Unit 1

Introduction of Database

- Purpose of database system
- View of data
- Database languages
- Transaction management
- Database administrator
- System structure

Data:

Data is commonly defined as raw facts or observation, typically about physical phenomena or business transactions. For example of data would be the marks obtained by students in different subjects. Data can be in any form-numerical, textual, graphical, image, sound, video etc.

The following figure shows the hierarchy of data storage.

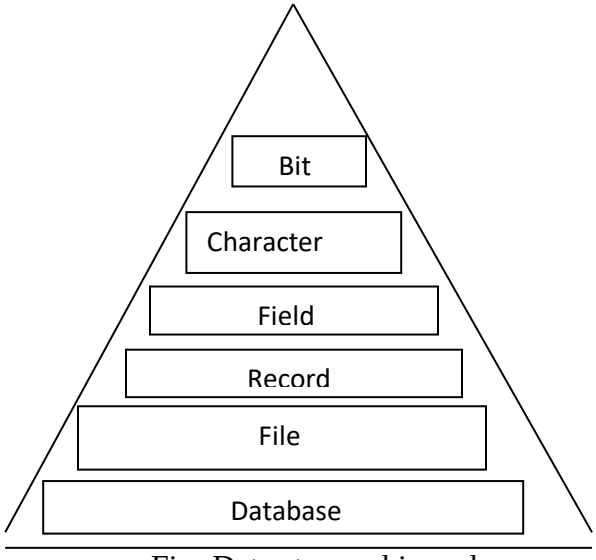


Fig:-Data storage hierarchy

Information:

Information is defined as refined or processed data that has been transformed into meaningful and useful form for specific users. For example, after processing the marks obtained by student it transformed into information, which is meaningful and from which we can decide which student stood first, second and so forth. Information comes from data and takes the form of table, graphs, diagrams etc.

Database:

A **database-management system** (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to as the **database**, contains information relevant to an enterprise. The primary goal of a DBMS is to provide a way to store and retrieve database information that is both *convenient* and *efficient*.

Database systems are designed to manage large bodies of information. Management of data involves both defining structures for storage of information and providing mechanisms for the manipulation of information. In addition, the database system must ensure the safety of the information stored, despite system crashes or attempts at unauthorized access. If data are to be shared among several users, the system must avoid possible anomalous results.

Database System Applications

Databases are widely used. Here are some representative applications:

- *Banking*: For customer information, accounts, and loans, and banking transactions.
- *Airlines*: For reservations and schedule information. Airlines were among the first to use databases in a geographically distributed manner—terminals situated around the world accessed the central database system through phone lines and other data networks.
- *Universities*: For student information, course registrations, and grades. *Credit card transactions*: For purchases on credit cards and generation of monthly statements.
- *Telecommunication*: For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.

- *Finance*: For storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.
- *Sales*: For customer, product, and purchase information.
- *Manufacturing*: For management of supply chain and for tracking production of items in factories, inventories of items in warehouses/stores, and orders for items.
- *Human resources*: For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.

STUDENT

Name	Student_number	Class	Major
Smith	17	1	CS
Brown	8	2	CS

COURSE

Course_name	Course_number	Credit_hours	Department
Intro to Computer Science	CS1310	4	CS
Data Structures	CS3320	4	CS
Discrete Mathematics	MATH2410	3	MATH
Database	CS3380	3	CS

SECTION

Section_identifier	Course_number	Semester	Year	Instructor
85	MATH2410	Fall	07	King
92	CS1310	Fall	07	Anderson
102	CS3320	Spring	08	Knuth
112	MATH2410	Fall	08	Chang
119	CS1310	Fall	08	Anderson
135	CS3380	Fall	08	Stone

GRADE_REPORT

Student_number	Section_identifier	Grade
17	112	B
17	119	C
8	85	A
8	92	A
8	102	B
8	135	A

PREREQUISITE

Course_number	Prerequisite_number
CS3380	CS3320
CS3380	MATH2410
CS3320	CS1310

Figure 1.2
A database that stores student and course information.

TRANSCRIPT

(a)

Student_name	Student_transcript				
	Course_number	Grade	Semester	Year	Section_id
Smith	CS1310	C	Fall	08	119
	MATH2410	B	Fall	08	112
Brown	MATH2410	A	Fall	07	85
	CS1310	A	Fall	07	92
	CS3320	B	Spring	08	102
	CS3380	A	Fall	08	135

COURSE_PREREQUISITES

(b)

Course_name	Course_number	Prerequisites
Database	CS3380	CS3320
		MATH2410
Data Structures	CS3320	CS1310

Figure 1.5
Two views derived from the database in Figure 1.2. (a) The TRANSCRIPT view. (b) The COURSE_PREREQUISITES view.

Figure 1.6
Redundant storage of Student_name and Course_name in GRADE_REPORT. (a) Consistent data. (b) Inconsistent record.

(a)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Smith	112	MATH2410	B
17	Smith	119	CS1310	C
8	Brown	85	MATH2410	A
8	Brown	92	CS1310	A
8	Brown	102	CS3320	B
8	Brown	135	CS3380	A

(b)

GRADE_REPORT

Student_number	Student_name	Section_identifier	Course_number	Grade
17	Brown	112	MATH2410	B

Database Systems versus File Systems

Consider part of a savings-bank enterprise that keeps information about all customers and savings accounts. One way to keep the information on a computer is to store it in operating system files. To allow users to manipulate the information, the- system has a number of application programs that manipulate the files, including

- A program to debit or credit an account
- A program to add a new account
- A program to find the balance of an account
- A program to generate monthly statements

file-processing system is supported by a conventional operating system.The system stores permanent records in various files, and it needs different application programs to extract records from, and add records to, the appropriate files. Before database management systems (DBMSs) came along, organizations usually stored information in such systems.

Keeping organizational information in a file-processing system has a number of major disadvantages:

- **Data redundancy and inconsistency**. Since different programmers create the files and application programs over a long period, the various files are likely to have different formats and the programs may be written in several programming languages. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular

customer may appear in a file that consists of savings-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. In addition, it may lead to **data inconsistency**; that is, the various copies of the same data may no longer agree. For example, a changed customer address may be reflected in savings-account records but not elsewhere in the system.

Difficulty in accessing data. Suppose that one of the bank officers needs to find out the names of all customers who live within a particular postal-code area. The officer asks

the data-processing department to generate such a list. Because the designers of the original system did not anticipate this request, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: either obtain the list of all customers and extract the needed information manually or ask a system programmer to write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is written, and that, several days later, the same officer needs to trim that list to include only those customers who have an account balance of $10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

Data isolation. Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

- **Integrity problems**. The data values stored in the database must satisfy certain types of **consistency constraints**. For example, the balance of a bank account may never fall below a prescribed amount (say, $25). Developers enforce these constraints in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

- **Atomicity problems**. A computer system, like any other mechanical or electrical device, is subject to failure. In many applications, it is crucial that, if a failure occurs, the data be restored to the consistent state that existed prior to the failure. Consider a program to transfer $50 from account *A* to account *B*. If a system failure occurs during the execution of the program, it is possible that the $50 was removed from account *A* but was not credited to account *B*, resulting in an inconsistent database state. Clearly, it is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic*—it must happen in its entirety or not at all. It is difficult to ensure atomicity in a conventional file-processing system.

- **Concurrent-access anomalies**. For the sake of overall performance of the system and faster response, many systems allow multiple users to update the data simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data. Consider bank account *A*, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. Suppose that the programs executing on behalf of each withdrawal read the old balance, reduce that value by the amount being withdrawn, and write the result back. If the two programs run concurrently, they may both read the value $500, and write back $450 and $400, respectively. Depending on which one writes the value last, the account may contain either $450 or $400, rather than the correct value of $350. To guard against this possibility, the system must maintain some form of supervision. But

supervision is difficult to provide because data may be accessed by many different application programs that have not been coordinated previously.

• **Security problems**. Not every user of the database system should be able to access all the data. For example, in a banking system, payroll personnel need to see only that part of the database that has information about the various bank employees. They do not need access to information about customer accounts. But, since application programs are added to the system in an ad hoc manner, enforcing such security constraints is difficult.

Characteristics of data in database:

The data in a database should have the following features:

- **Shared:** Data should be sharable among different users and applications.
- **Persistence:** Data should exist permanently in the database. Changes in the database must not be lost because of any failure.
- **Validity/Integrity/Correctness:** It should maintain the integrity so that there is always correct data in the database.
- **Security:** Data should be protected from unauthorized access.
- **Non-redundancy:** Data should not be repeated.
- **Consistency:** A consistent state of the database satisfies all the constraints specified in the database. Data in a database is consistent if any changes in the database take the database from one consistent state to another.
- **Independence:** The three levels in the schema should be independent of each other so that the changes in the schema in one level should not affect the other levels.

Purpose of DBMS (Functions of DBMS):

The benefits of using DBMS are:

- **To reduce redundancy:**Repeating of the same information in database is called redundancy of data which leads to several problems such as wastage of space, duplication effort for entering data and inconsistency. When DBMS is used and database is created, redundancy is minimized.
- **To avoid inconsistency:**The database is said to be inconsistent if various copies of the same data may no longer agree. For example, a changed customer address may be reflected in saving account but not elsewhere in the system. By using DBMS we can avoid inconsistency.
- **To share data:**The data in the database can be shared among many users and applications. The data requirements of new applications may be satisfied without having to create any new stored files.

- **To provide support for transactions:** A transaction is a sequence of database operations that represents a logical unit of work. It accesses a database and transforms it from one state to another. A transaction can update a record, delete one, modify a set of records etc. when the DBMS does a 'commit', the changes made by the transaction are made permanent. We can roll back the transaction to undo the effects of transaction.
- **To maintain integrity:**Most database applications have certain integrity constraints that must hold for the data. A DBMS provides capabilities for defining and enforcing these constraints. For example, the value of roll number field of each student in student database should be unique for each student. It is a type of rule. Such a rule is enforced using constraint at the time of creation of database.
- **To enforce security:** Not every user of the database system should be able to access all data.Different checks can be established for each type of access (retrieve, modify, delete, etc) toeach piece of information in the database.
- **To provide efficient backup and recovery:** Provide facilities for recovering from software and hardware failures to restore database to previous consistent state.
- **To Concurrent Access Database:** Concurrent access means access to the same data simultaneously by more than one user. The same data may be used by many users for the purpose reading at the same time. But when a user tries to modify a data, there should be a concurrency control mechanism to avoid the inconsistency of data. A DBMS provides facilities for these operations.

Disadvantages of DBMS

- **Problem associated with centralization:** Centralization increases the security problems.
- **Cost of software:** Today's there are several softwares which are very costly. Hence from economic point of view it is the drawback.
- **Cost of hardware:**To support various software some upgraded hardware components are needed.Hence from economic point of view it is the drawback.
- **Complexity of backup and recovery:**DBMS provides the centralization of the data, which requires the adequate backups of data.
- Overhead for providing generality, security, recovery, integrity, and concurrency control.
- If the database and applications are simple, well defined, and not expected to change.
- If there are stringent real-time requirements that may not be met because of DBMS overhead.

Differences bet^{n} DBMS and file processing system:

DBMS	File processing system
1. A Database Management System (DBMS) is a collection of interrelated data and the set of programs to access those data. 2. Data redundancy problem is not found. 3. Data inconsistency does not exist. 4. Accessing data from database is easier. 5. The problem of data isolations is not found. 6. Atomicity and integrating problems are not found. 7. Data are more secure. 8. Concurrent access and crash recovery.	1. A flat file system stores data in a plain text file.A flat file is a file that contains records, and in which each record is specified in a single line. 2. Data redundancy problem exist. 3. Data inconsistency may exist. 4. Accessing data from database is comparatively difficult. 5. Here data are scattered in various files and formats so data isolation problem exist. 6. Here these problems are found. 7. Data are less secure. 8. Here there is no concurrent access and no recovery.

Views of Data/ Data abstraction:

The system hides certain details of how the data are stored and maintained and such view is an abstract view.

✓ The Database System provides users with an *abstract view* of the data.

Data Abstraction:-*The database designers use the complex data structure to represent the data in the database and developer hides the complexity from user from several level of abstraction such as physical level, logical level, and view level. This process is called data abstraction*.

Levels of Data Abstraction:

The three levels of data abstraction can be shown as follows:

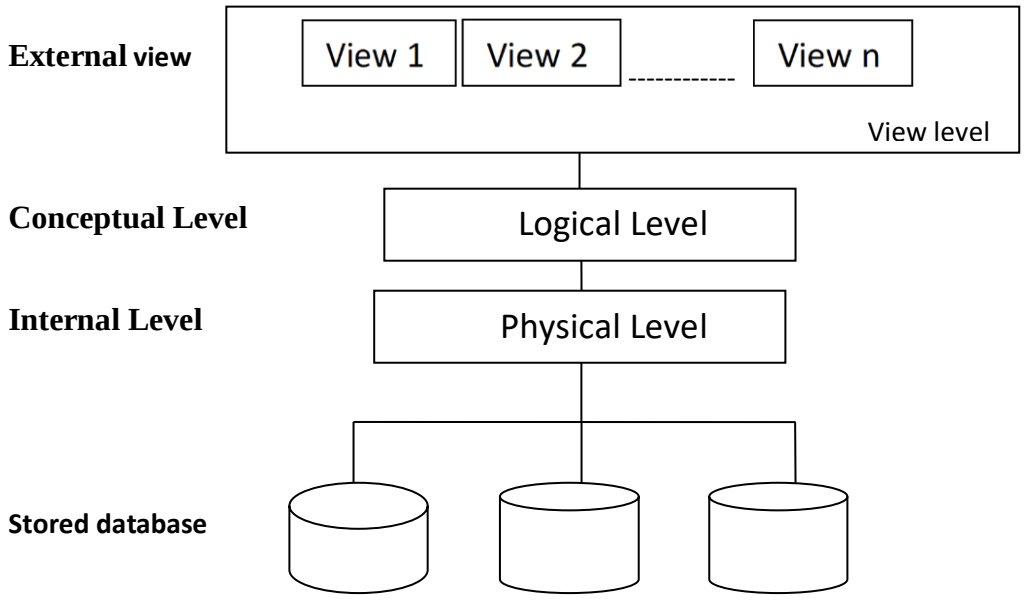


Fig: Three levels of data abstraction

Physical level

- ✓ It is the lowest level of abstractions describes how the data are actually stored.
- ✓ The physical level describes complex low level data structure in details.
- ✓ At this level records such as customer, account can be described as a block of consecutive storage location (e.g. byte, word)
- ✓ The database system hides many of the lowest level storage details from database programmer. Database administrator may be aware of certain details of the physical organization of the data.

Logical level

- ✓ It is the next higher level of data abstraction which describes what data are stored in the database, and what relationships exist among those data.
- ✓ At the logical level , each record is described by a type definition
- ✓ Programmers and database administrator works at this level of abstraction.

View level

- ✓ It is the highest level of abstraction describes only a part of the database and hides some information to the user.
- ✓ At view level, computer users see a set of application programs that hide details of data types. Similarly at the view level several views of the database are defined and database user see only these views.

- ✓ Views also provides the security mechanism to prevent users from accessing certain parts of the database (that is views can also hide information (such as an employee‘s salary) for security purposes.)

Instances and Schemas

Instance (Database State):

The collection of information stored in the database at a particular moment is called an instance of the database. It is the actual content of the database at a particular point in time

✓ The term *instance* is also applied to individual database components, e.g. *record instance, table instance, entity instance.*

Initial Database State

Refers to the database state when it is initially loaded into the system.

Valid State

A state that satisfies the structure and constraints of the database.

Schema:

The overall design of the database which is not expected to change frequently is called database schema. Simply, the database schema is the logical structure of the database.

- ✓ The concept of database schema and instances can be understood by analogy to a program written in a programming language
- ✓ A database schema corresponds to the variable declaration and the values of the variables in a program at a point in time correspond to an instance of a database.
- ✓ The database systems have several schemas and partitioned according to the level of abstraction such as physical and logical schema.

STUDENT

Name	Student-number	Class	Major

Fig: Schema diagram for Student

Note:

- ✓ The ***database schema*** changes very infrequently.
- ✓ The ***database state*** changes every time the database is updated.

Data Independence:

The three schema architecture further explains the concept of data independence, the capacity to change the schema at one level without having to change the schema at the next higher level.

- ➢ **Logical Data Independence**

- **Physical Data Independence**

Logical Data Independence:
The capacity to change the conceptual schema without having to change the external schemas and their associated application programs is called logical data independence. When modification is done to the conceptual schema (i.e tables) the mapping called "external mapping" is changes automatically by DBMS.

Physical Data Independence:
The capacity to change the internal schema without having to change the conceptual schema is called physical data independence.When a schema at a lower level is changed, only the **mappings** between this schema and higher-level schemas need to be changed in a DBMS. This mapping is called "logical mapping".
For example, the internal schema may be changed when certain file structures are reorganized or new indexes are created to improve database performance.

Database Languages:
DBMS provides two languages: Data-Definition Language (DDL) and Data-Manipulation Language (DML).

Data Definition Language (DDL):
Data definition language is the specification notation for defining the database schema.

- ✓ Used by the DBA and database designers to specify the conceptual schema of a database.
- ✓ In many DBMSs, the DDL is also used to define internal and external schemas (views).

Example:

```
CREATE TABLE account
(
        account-number      CHAR(10),
        balance             INTEGER
)
```

✓ Execution of the above DDL statement creates the account table.
✓ It updates a special set of tables called the data dictionary.

Data dictionary: DDL compiler generates a set of tables stored in a *data dictionary*. *Simply*, **Data dictionary** ***is a special set of tables that contain the information about tables.*** Data dictionary contains metadata (i.e., data about data)

- **Metadata**: ***data that describes the database or one of its parts is called metadata. The schema of a table is an example of metadata***

The **DDL provides** the facilities to define:

- ✓ **Database scheme**
- ✓ **Database tables**

- ✓ **Integrity constraints**
  - Domain constraints
  - Referential integrity (references constraint in SQL)
  - Assertions
  - Triggers
  - Views
- ✓ **Security and Authorization**
- ✓ **Modify the Scheme**

The common DDL Commands are: CREATE, ALTER,DROP

Data Manipulation Language (DML):

A **Data-manipulation language** (DML***) is a language that enables users to access or manipulate data organized by the appropriate data model***.DML also known as query language.

Ex.
```
SELECT *
FROM account
WHERE balance<1000
```
Execution of this statement retrieves the records of all accounts in which balance is below 1000.

There are basically two classes of DML:

Procedural DMLs (or Low-level DML):

In procedural DMLs, a user specifies what data are required and how to get those data.

Declarative (or nonprocedural or high-level) DMLs:

In declarative DMLs a user specifies what data are needed without specifying how to get those data

The data manipulation is:

- ✓ The retrieval of information stored in the database
- ✓ The insertion of new information into the database
- ✓ The deletion of information from the database
- ✓ The modification of information stored in the database

Thecommon DML commands:SELECT, INSERT, UPDATE, DELETE

Query: ***A query is a statement requesting the retrieval of information***. SQL is the most widely used query language Select, insert, update, delete etc are the SQL DML statement

Data Manipulation Language:

By Data Manipulation, we mean

- ➢ The retrieval of information stored in the database.
- ➢ The insertion of new information into the database.

- The deletion of information from the database.
- The modification of information stored in the database.

There are basically two types of DML:

- ✓ **Procedural DMLs**: require the user to specify what data are needed and how to get those data.
- ✓ **Non-Procedural DMLs**: require a user to specify what data are needed without specifying how to get those data.

Transaction Management:

Collections of operations that form a single logical unit of work are called transactions. For example, a transfer of funds from a checking account to a savings account is a single operation from the customer's standpoint; within the database system, however, it consists of several operations.A database system must ensure proper execution of transactions despite failures—either the entire transaction executes, or none of it does.

Consider the transaction,

```
A = A – 50;
Write (A);
Read (B);
B = B+50;
Write (B);
```

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

- ✓ **Atomicity:** Either all operations of the transaction are reflected properly in the database or none are. Suppose during the execution of above transaction a failure occurred after the write (A) operation but before write (B) operation. Then the values of amount reflected in database will be 950 and 2000. The system destroyed 50 as a result of failure.

- ✓ **Consistency:** Database must be in correct state before and after execution of the transaction. The consistency requirement here is that sum of A and B be unchanged by the execution of transaction. Without the consistency requirement, money could be created or destroyed by a transaction.

- ✓ **Isolation:** Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions T_i and T_j it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.

- ✓ **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are called ACID properties, with acronym derived from the first letters of the above four properties.

Transactions access data using two operations:

- ✓ **Read(x):** Which transfers the data item x from the database to a local buffer belonging to the transaction that executed the read operation.
- ✓ **Write(x):** Which transfers the data item x from the local buffer of the transaction that executed to the database.

Database users and administrators

Database users:

Users are differentiated by the way they expect to interact withthe system. There are four different types of database-system users:

- ✓ **Naive Users**: They are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example: a bank teller who needs to transfer $50 from account A to account B invokes a program called transfer.
- ✓ **Application programmers:**Theyare computer professionals who write application programs. Application programmers can choose any tools to develop user interface.
- ✓ **Sophisticated users:**They interact with the system without writing programs. They form their requests in a database query language.
- ✓ **Specialized users**: They are sophisticated users who write specialized database applications that do not fit into the traditional data-processing framework. Among these applications are computer-aided design systems, knowledge base and expert systems, systems that store data with complex data types (graphics data and audio data), and environment-modeling systems.

Database Administrator:

The person who has such central control over the system is called the database administrator (DBA). The function of the DBA includes the following:

- ✓ **Schema definition**: The DBA creates the original database schema by writing a set of definitions that is translated by the DDL compiler to a set of tables that is stored permanently in the data dictionary.
- ✓ **Schema and physical-organization modification**: The DBA carries out changes to the schema and physical organization to reflect the changing needs to the organization, or to alter the physical organization to improve performance.
- ✓ **Granting the authorization of data access**: The granting of different types of privileges to the database users so that all the users are not able to all data.
- ✓ **Integrity-constraint specifications**: The data values stored in the database must satisfy certain consistency constraints. The database administrator must specify such a constraint explicitly.
- ✓ **Routine maintenance:** Routine maintenance includes periodic backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding, Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required etc.

Database Models:

Data model is a collection of tools for describing data, data relationships, data semantics and data constraints. The database model refers the way for organizing and structuring the data in

the database. Traditionally, there are different database models which are used to design and develop the database of the organization.

1. **Entity- Relationship Model**
2. **Object oriented Model**

Object based data model

3. **Relational Model**
4. **Hierarchical model**
5. **Network Model**
6. **Object Relational Data Model**

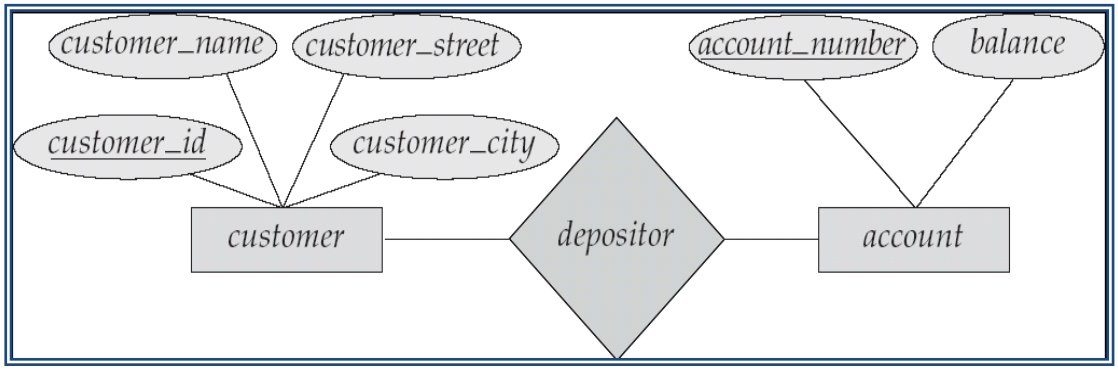
Record based data models

1. Entity- Relationship Model:

The E-R data models is based on a perception of real world that consist of a collection of basic objects called entities and relationship among these objects.In an E-R model a **database**can be modeled as a collection of ***entities, and relationship*** among entities.

Overall logical structure of a database can be expressed graphically by E-R diagram. The basic components of this diagram are:

- **Rectangles** (represent entity sets)
- **Ellipses** (represent attributes)
- **Diamonds** (represent relationship sets among entity sets)
- **Lines** (link attributes to entity sets and entity sets to relationship sets)



2. Relational Model:

It is the current favorite model. The relational model is a lower level model that uses a collection of tables to represent both data and relationships among those data. Each table has multiple columns, and each column has a unique name. Each table corresponds to an entity set or relationship set, and each row represents an instance of that entity set or relationship set. Structured query language (SQL) is used to manipulate data stored in tables.

Customer

Depositor (Relationship table)Account

Customer_id	Account_no
1	Ac-33
2	Ac-12
3	Ac-65
3	Ac-77
2	Ac-33

Account_no	Balance
Ac-33	10000
Ac-65	20000
Ac-12	70000
Ac-77	9000

3. Object oriented data model:

The object oriented data model is based on object-oriented programming paradigm. It is based on the concept of encapsulating the data and the functions that operate on those data in a single unit called an object. The internal parts of objects are not visible externally.
Here one object communicates with other objects by sending message.

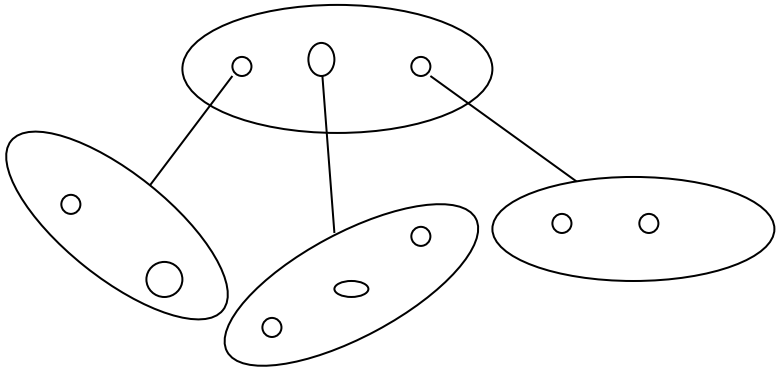


Fig: Object oriented data model

Customer_id	Customer_name	Customer_street	Customer_city
1	Bhupi	Chandani	Kanchanpur
2	Arjun	Balkhu	Kathmandu
3	Abin	Pulchoak	Lalitpur

4. Hierarchical data model:

In a hierarchical data model, the data elements are linked in the form of an inverted tree structure with the root at the top and the branches formed below. Below the single root data

element are subordinate elements, each of which, in turn, has one or more other elements. There is a parent child relationship among the data elements of a hierarchical database. There may be many child elements under each parent element, but there can be only one parent element for any child element. The branches in the tree are not connected.

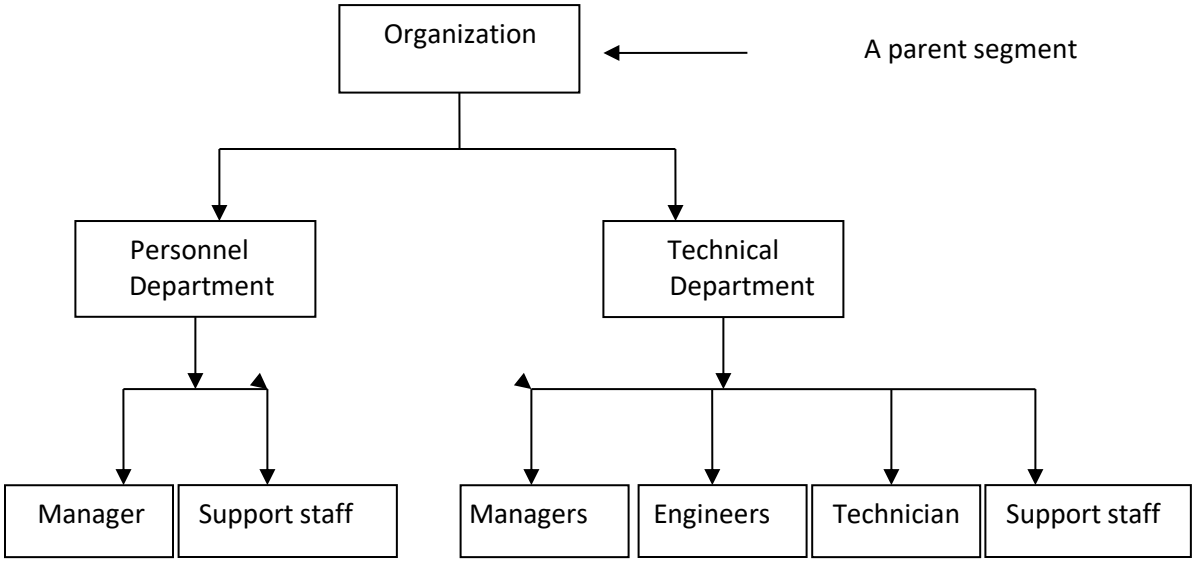
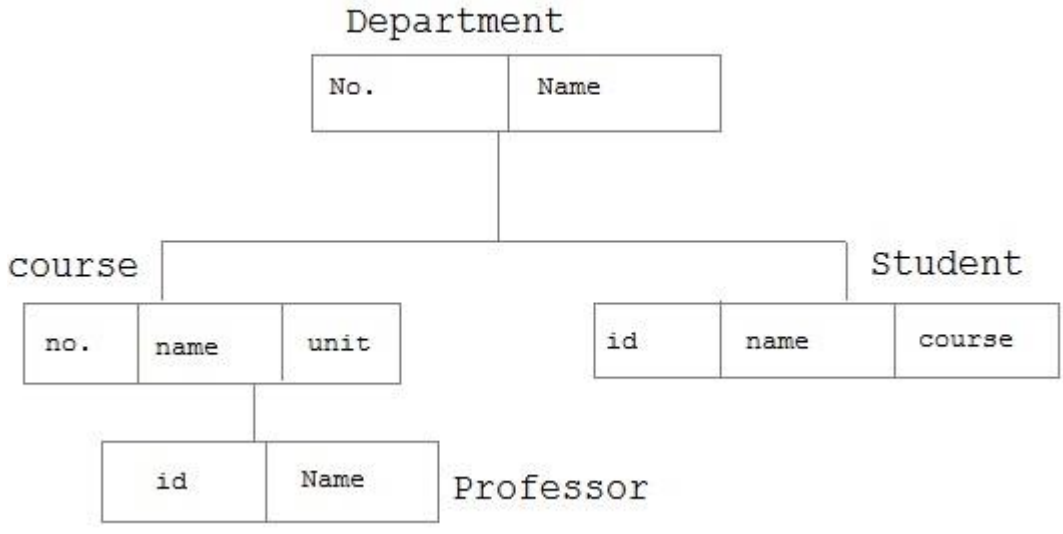


Fig: An example of hierarchical database model



The main limitation of this structure is that it does not support flexible data access, because data can be accessed only by following the path down the tree structure.

Advantages of hierarchical database model

- It is the easiest model of database.

- It is secure model as nobody can modify the child without consulting to its parent.
- Searching is fast.
- Very efficient in handling one- to- many relationship.

Disadvantages hierarchical database model

- It is old fashion, outdated database model
- Modification and addition of child without consulting its parent is impossible.
- Cannot handle many- to- many relationships.
- Increase redundancy.
- It does not support flexible data access, because data can be accessed only by following the path down the tree structure.

5. Network Data Model:

A network data model is an extension of the hierarchical database structure. In this model also, the data elements of a database are organized in the form of parent-child relationships and all types of relationships among the data elements must be determined when the database is first designed. In a network database, a child data element can have more than one parent element or no parent at all. Moreover, in this type of database, the database management system permitsthe extraction of the needed information from any data element in the database structure, instead of s3tarting from the root data element.

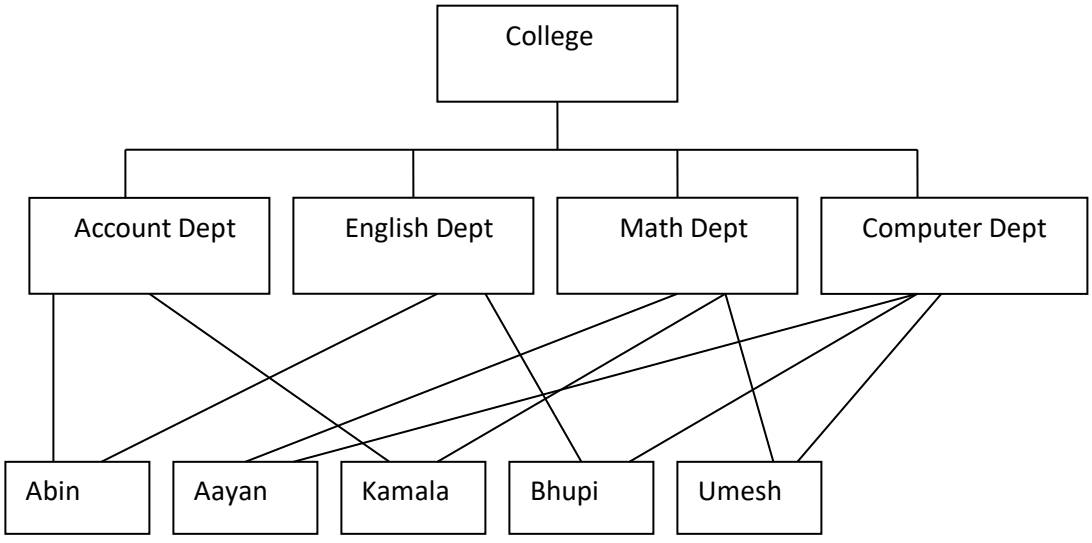


Fig: An example of a network database

Advantages of network database model

- More flexible than hierarchical database because it accept many to many relationship.
- Searching is faster because of multidirectional pointers.
- Promotes database integrity
- Data independence

Disadvantages of network database model

- Less secure than hierarchical as it is open to all.
- Need long program to handle the relationship.
- Pointer is used in this database and that increased the overhead of storages
- Lack of structural independence

Centralized and Client/Server Architectures for DBMSs

Centralized DBMSs Architecture

Architectures for DBMSs have followed trends similar to those for general computer system architectures. Earlier architectures used mainframe computers to provide the main processing for all system functions, including user application programs and user interface programs, as well as all the DBMS functionality. The reason was that most users accessed such systems via computer terminals that did not have processing power and only provided display capabilities. Therefore, all processing was performed remotely on the computer system, and only display information and controls were sent from the computer to the display terminals, which were con-nected to the central computer via various types of communications networks.

As prices of hardware declined, most users replaced their terminals with PCs and workstations. At first, database systems used these computers similarly to how they had used display terminals, so that the DBMS itself was still a **centralized** DBMS in which all the DBMS functionality, application program execution, and user inter-face processing were carried out on one machine. Figure 2.4 illustrates the physical components in a centralized architecture. Gradually, DBMS systems started to exploit the available processing power at the user side, which led to client/server DBMS architectures.

Basic Client/Server Architectures

First, we discuss client/server architecture in general, then we see how it is applied to DBMSs. The **client/server architecture** was developed to deal with computing envi-ronments in which a large number of PCs, workstations, file servers, printers, data-

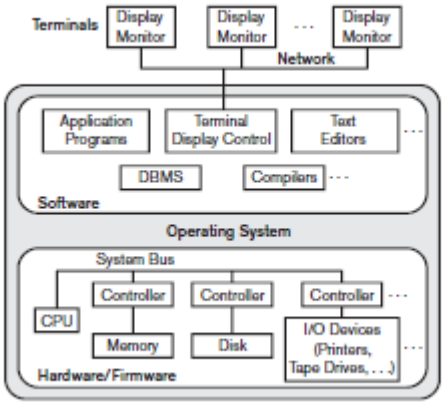


Figure 2.4
A physical centralized architecture.

base servers, Web servers, e-mail servers, and other software and equipment are connected via a network. The idea is to define **specialized servers** with specific functionalities. For example, it is possible to connect a number of PCs or small workstations as clients to a **file server** that maintains the files of the client machines. Another machine can be designated as a **printer server** by being connected to vari-ous printers; all print requests by the clients are forwarded to this machine. **Web servers** or **e-mail servers** also fall into the specialized server category. The resources provided by specialized servers can be accessed by many client machines. The **client machines** provide the user with the appropriate interfaces to utilize these servers, as well as with local processing power to run local applications. This concept can be carried over to other software packages, with specialized programs—such as a CAD (computer-aided design) package—being stored on specific server machines and being made accessible to multiple clients. Figure 2.5 illustrates client/server archi-tecture at the logical level; Figure 2.6 is a simplified diagram that shows the physical architecture. Some machines would be client sites only (for example, diskless work-stations or workstations/PCs with disks that have only client

software installed).

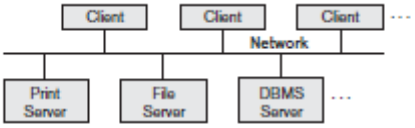


Figure 2.5
Logical two-tier client/server architecture.

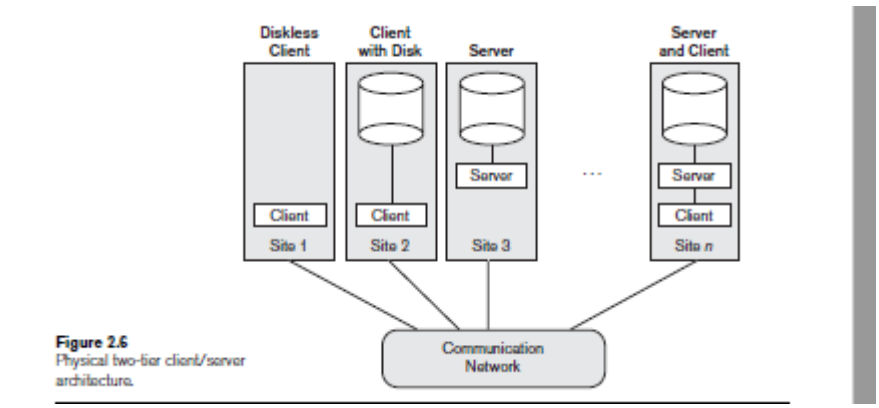


Figure 2.6
Physical two-tier client/server architecture.

Other machines would be dedicated servers, and others would have both client and server functionality.

The concept of client/server architecture assumes an underlying framework that consists of many PCs and workstations as well as a smaller number of mainframe machines, connected via LANs and other types of computer networks. A **client** in this framework is typically a user machine that provides user interface capabilities and local processing. When a client requires access to additional functionality— such as database access—that does not exist at that machine, it connects to a server that provides the needed functionality. A **server** is a system containing both hard-ware and software that can provide services to the client machines, such as file access, printing, archiving, or database access. In general, some machines install only client software, others only server software, and still others may include both client and server software, as illustrated in Figure 2.6. However, it is more common that client and server software usually run on separate machines. Two main types of basic DBMS architectures were created on this underlying client/server framework: **two-tier** and **three-tier**.[13] We discuss them next.

Two-Tier Client/Server Architectures for DBMSs

In relational database management systems (RDBMSs), many of which started as centralized systems, the system components that were first moved to the client side were the user interface and application programs. Because SQL (see Chapters 4 and 5) provided a standard language for RDBMSs, this created a logical dividing point between client and server. Hence, the query and transaction functionality related to SQL processing remained on the server side. In such an architecture, the server is often called a **query server** or **transaction server** because it provides these two functionalities. In an RDBMS, the server is also often called an **SQL server**.

The user interface programs and application programs can run on the client side. When DBMS access is required, the program establishes a connection to the DBMS (which is on the server side); once the connection is created, the client program can communicate with the DBMS. A standard called **Open Database Connectivity** (**ODBC**) provides an **application programming interface** (**API**), which allows client-side programs to call the DBMS, as long as both client and server machines have the necessary software installed. Most DBMS vendors provide ODBC drivers for their systems. A client program can actually connect to several RDBMSs and send query and transaction requests using the ODBC API, which are then processed at the server sites. Any query results are sent back to the client program, which can process and display the results as needed. A related

standard for the Java program-ming language, called **JDBC**, has also been defined. This allows Java client programs to access one or more DBMSs through a standard interface.

The different approach to two-tier client/server architecture was taken by some object-oriented DBMSs, where the software modules of the DBMS were divided between client and server in a more integrated way. For example, the **server level** may include the part of the DBMS software responsible for handling data storage on disk pages, local concurrency control and recovery, buffering and caching of disk pages, and other such functions. Meanwhile, the **client level** may handle the user interface; data dictionary functions; DBMS interactions with programming lan-guage compilers; global query optimization, concurrency control, and recovery across multiple servers; structuring of complex objects from the data in the buffers; and other such functions. In this approach, the client/server interaction is more tightly coupled and is done internally by the DBMS modules—some of which reside on the client and some on the server—rather than by the users/programmers. The exact division of functionality can vary from system to system. In such a client/server architecture, the server has been called a **data server** because it pro-vides data in disk pages to the client. This data can then be structured into objects for the client programs by the client-side DBMS software.

The architectures described here are called **two-tier architectures** because the soft-ware components are distributed over two systems: client and server. The advan-tages of this architecture are its simplicity and seamless compatibility with existing systems. The emergence of the Web changed the roles of clients and servers, leading to the three-tier architecture.

Three-Tier and n-Tier Architectures for Web Applications

Many Web applications use an architecture called the **three-tier architecture**, which adds an intermediate layer between the client and the database server, as illustrated in Figure 2.7(a).

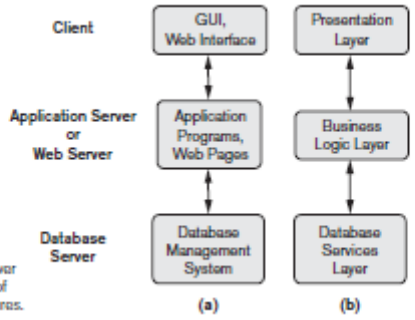


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

This intermediate layer or **middle tier** is called the **application server** or the **Web server**, depending on the application. This server plays an intermediary role by run-ning application programs and storing business rules (procedures or constraints) that are used to access data from the database server. It can also improve database security by checking a client's credentials before forwarding a request to the data-base server. Clients contain GUI interfaces and some additional application-specific business rules. The intermediate server accepts requests from the client, processes the request and sends database queries and commands to the database server, and then acts as a conduit for passing (partially) processed data from the database server to the clients, where it may be processed further and filtered to be presented to users in GUI format. Thus, the *user interface*, *application rules*, and *data access* act as the three tiers. Figure 2.7(b) shows another architecture used by database and other application package vendors. The presentation layer displays information to the user and allows data entry. The business logic layer handles intermediate rules and constraints before data is passed up to the user or down to the DBMS. The bottom layer includes all data management services. The middle layer can also act as a Web server, which retrieves query results from the database server and formats them into dynamic Web pages that are viewed by the Web browser at the client side.

Other architectures have also been proposed. It is possible to divide the layers between the user and the stored data further into finer components, thereby giving rise to *n*-tier architectures, where *n* may be four or five tiers. Typically, the business logic layer is divided into multiple layers. Besides distributing programming and data throughout a network, *n*-tier applications afford the advantage that any one tier can run on an appropriate processor or operating system platform and can

be handled independently. Vendors of ERP (enterprise resource planning) and CRM (customer relationship management) packages often use a *middleware layer*, which accounts for the front-end modules (clients) communicating with a number of back-end databases (servers).

Advances in encryption and decryption technology make it safer to transfer sensi-tive data from server to client in encrypted form, where it will be decrypted. The lat-ter can be done by the hardware or by advanced software. This technology gives higher levels of data security, but the network security issues remain a major con-cern. Various technologies for data compression also help to transfer large amounts of data from servers to clients over wired and wireless networks.

Unit 2

Entity-Relationship Model

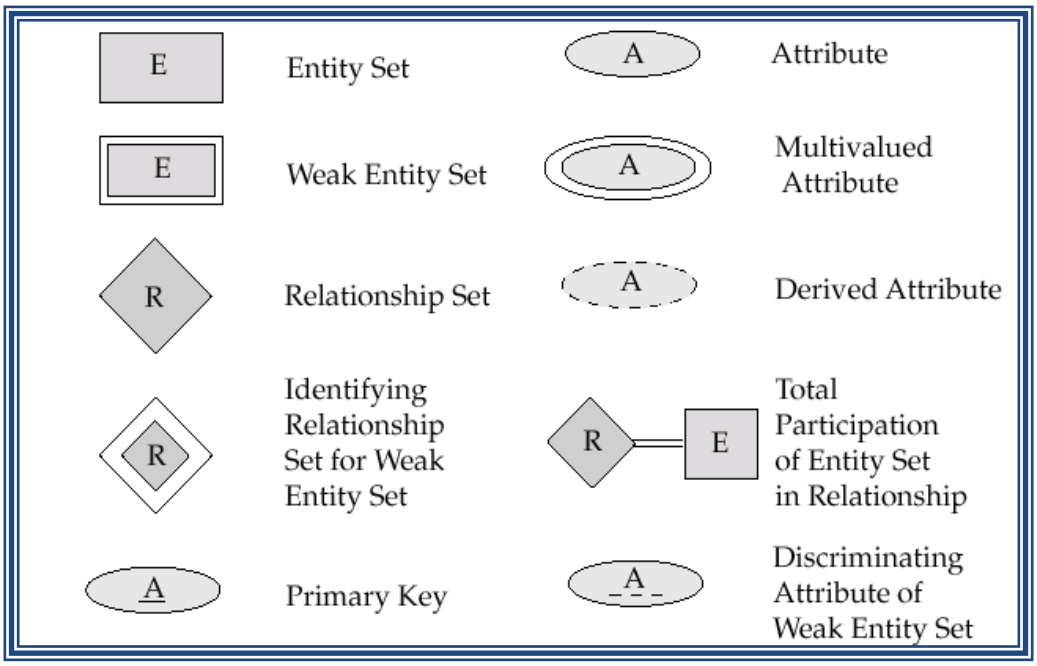
- Basic concepts
- Mapping constraints
- Key
- Entity-relationship diagram
- Weak entity set
- Extended E-R features
- Reduction of an E-R schema to tables

The E-R data models is based on a perception of real world that consist of a collection of basic objects called entities and relationship among these objects.In an E-R model a **database**can be modeled as a collection of ***entities, and relationship*** among entities.

Notation of E-R diagram:

- ✓ **Rectangles** represent entity sets.
- ✓ **Diamonds** represent relationship sets.
- ✓ **Lines** link attributes to entity sets and entity sets to relationship sets.
- ✓ **Ellipses** represent attributes
- ✓ **Double ellipses** represent multivalued attributes.
- ✓ **Dashed ellipses** denote derived attributes.
- ✓ **Underline** indicates primary key attributes
- ✓ **Double Lines** indicate total participation of an entity set in a relationship set.

- ✓ **Double Rectangles** represent weak entity sets.
- ✓ **Double Diamonds** represent identifying relationship set for weak entity set.



Entity:

An **entity** is an object that exists and is distinguishable from other objects. An entity may be has a set of properties and the values for some set of properties may uniquely identifies an entity. For example: specific person, specific company, event, a particular plant etc. The entities have *attributes*, *for e*xample people have *names* and *addresses.*

Eg: A particular person

Entity Set:

An **entity set** is a set of entities of the same type that share the same properties or attributes. For example: set of all persons who are customers at a particular Bank can be defined as the entity set customer.

Customer_id	Customer_name	Customer_street	Customer_city
1	Abhi	Simraungadh	Bara
2	Arjun	Balkhu	Kathmandu
3	Ashok	Pulchoak	Lalitpur

Eg:

Customer

Entity set

Entity

Attributes:

The properties or characteristics of an entity are called**attributes**. For example, a *customer* entity can have *customer-id*, *customer-name*, *customer-street*, and *customer-city* as attributes.

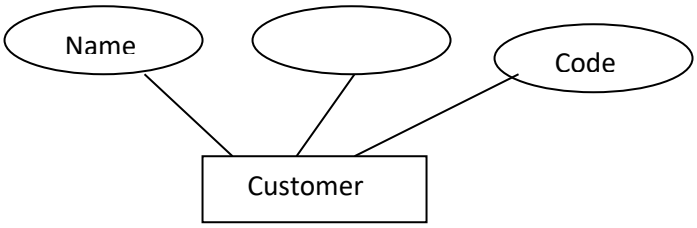


Fig: Entity with attributes

Attribute Types

An Attribute, as used in E-R model, can be characterized by the following attributes types:

Simple and composite attribute:

- **Simple:** The attributes that cannot be divided into subparts (ie. Into attributes) are called simple attributes. For example roll-number attribute of a student cannot be further divided into sub parts thus roll-number attribute of a student entity acts as a simple attribute.
- **Composite:** The attributes that can be divided into subparts (ie into attributes) are called composite attributes. For example name attribute of a particular student can be further vided into subparts first-name, middle-name, and last-name thus name attribute acts as a composite

.

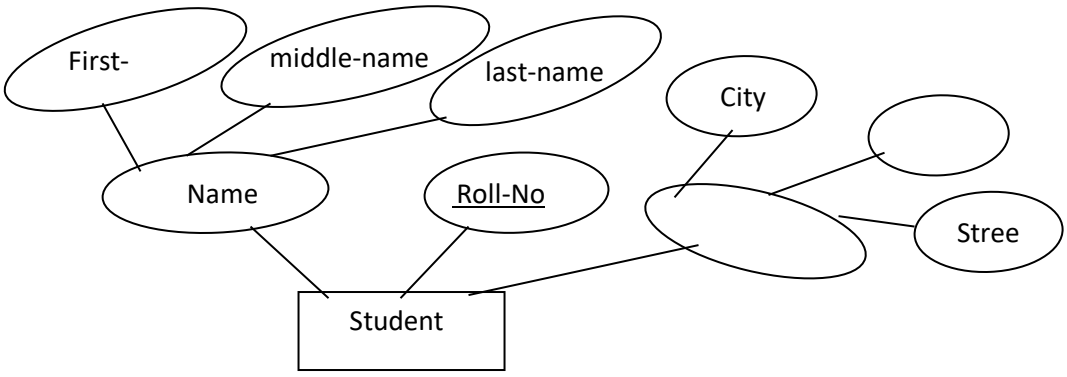
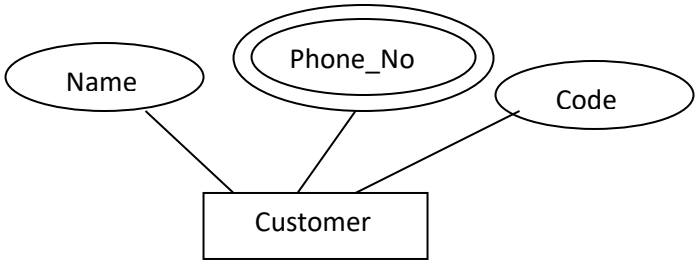


Fig:- Simple and composite attributes of Student entity

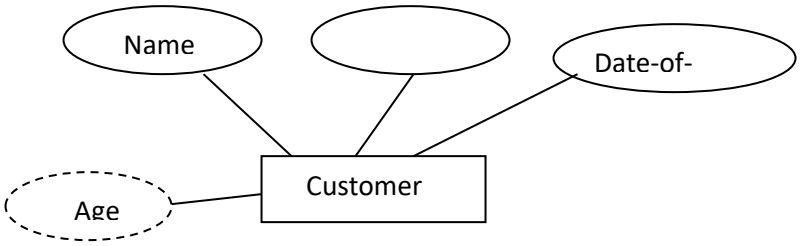
Single-valued and multi-valued attributes:

- **Single-valued:** The attributes which has a single value for a particular entity is called single-valued attributes. For example almost of our example has the single value attributes; loan-number specifies loan entity refers only one lone number.

- **Multi-valued:** If an attribute has a set of value for a specific entity is called multi-valued attributes. For example: multi-valued attribute: 'phone_number' of an employeemay have zero, one or several phone numbers.

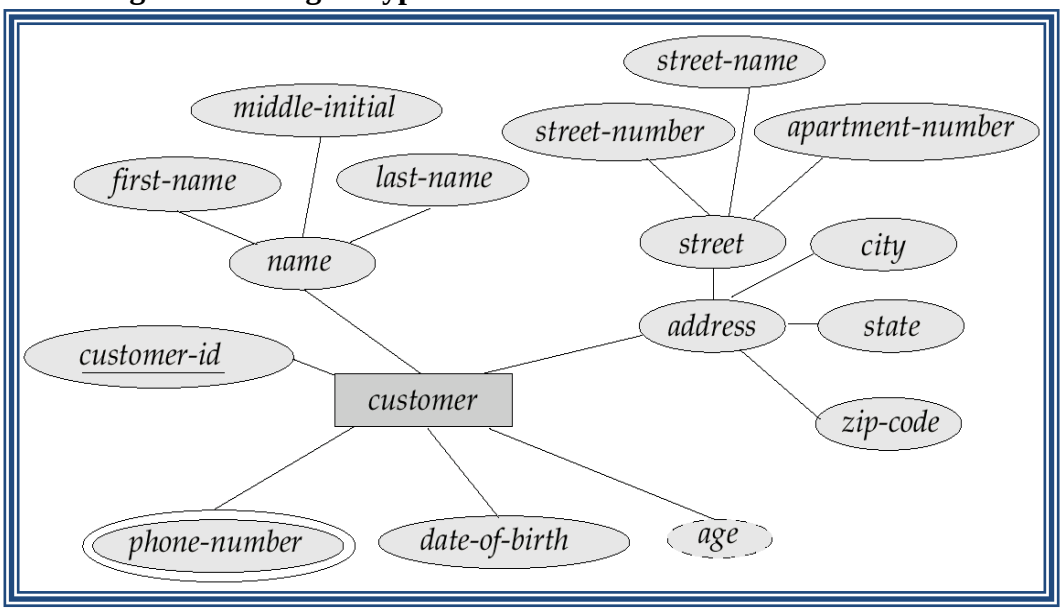


- **Derived attributes:**
  The attribute whose value derived from the values of other related attributes or entities is called derived attribute. For example: age, given date_of_birth.



Note: All attributes take a **null** value when an entity does not have a value for it. The null value may indicate "not applicable", that is, the value does not exist for the entity.

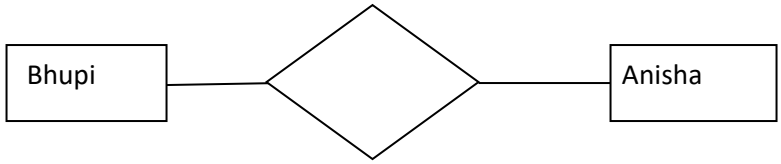
Example: E-R diagram showing all types of attribute



Relationship and Relationship sets:

A **relationship** is an association among two or more entities.

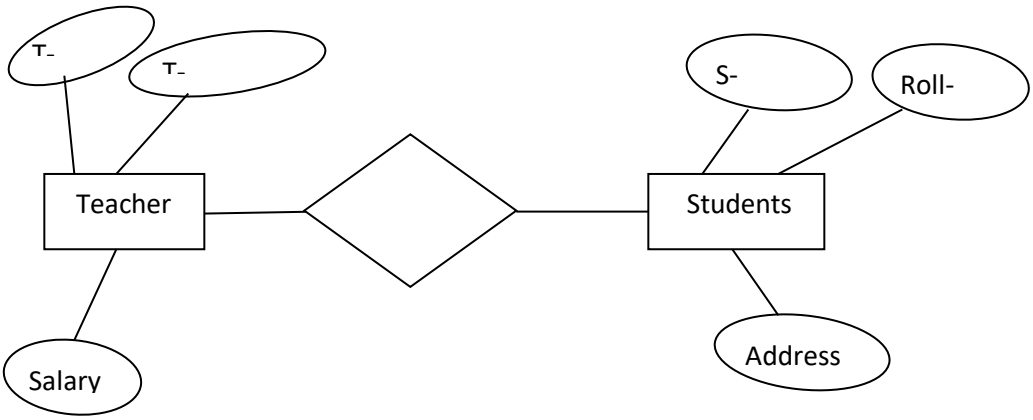
For example we may define a relationship which associates the teacher "Bhupi" with a student of name "Anisha". This specifies that Bhupi is a teacher who teaches a student of name "Anisha".



A **relationship set** is a set of relationships of the same type.

Formally, it is a mathematical relation on $n \geq 2$ entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of $\{(e_1, e_2, \ldots, e_n) | e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$, where $(e_1, e_2, \ldots, e_n)$ is a relationship.

For example: Teacher teaches students



A relationship set may also have attributes called ***descriptive attributes***. For example, the *teach* relationship set between entity sets *teacher*and *students*may have the attribute *teaches-date*.

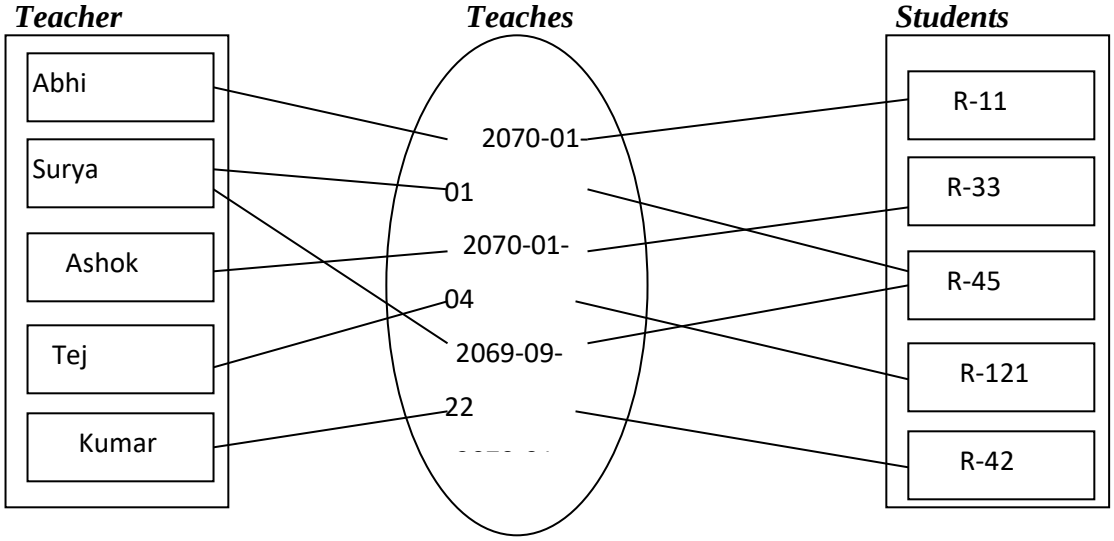
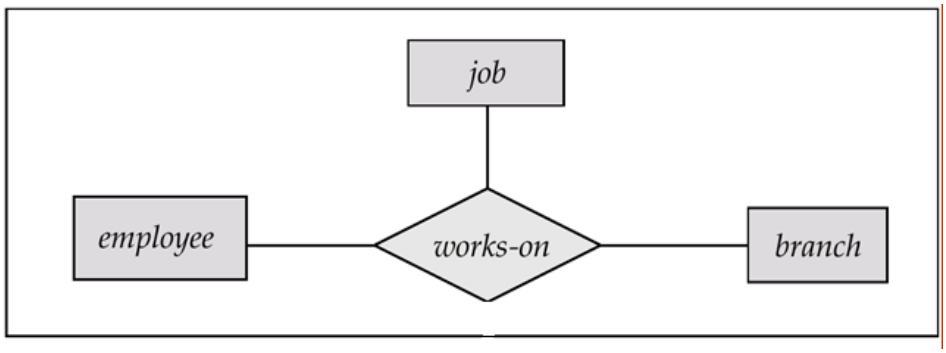


Fig: Showing relationships with descriptive attributes

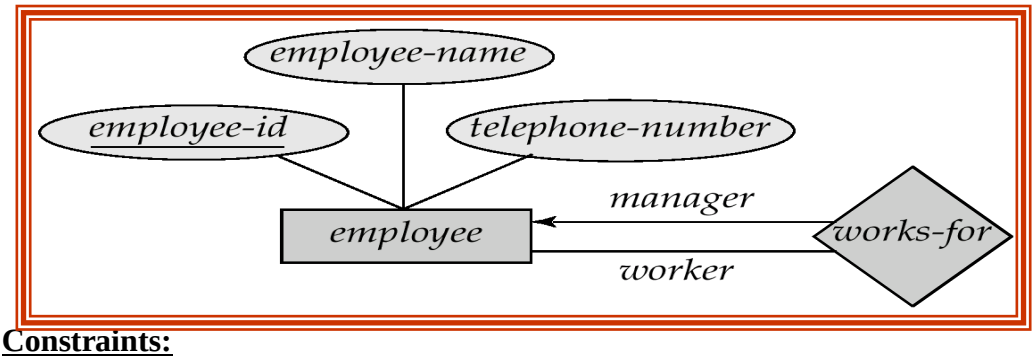
Degree of a relationship:

Degree of a relationship set refers to the number of entity sets that participate in a relationship set. Relationship sets that involve two entity sets are called **binary relationship** sets. Most relationship sets in a database system are binary.

Relationship sets may involve more than two entity sets called **n-ary** relationship sets but are rarer. For example, suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets employee, job and branch.



When the entity sets of a relationship set are not distinct (ie. The same entity set participates in a relationship set more than once, in different roles). This type of relationship set is sometimes called a **recursive** relationship set.



Constraints:

An entity relationship model may define certain constraints to which the contents of a database must conform. The most important constraints are: ***mapping cardinalities*** and ***participation constraints.***

1. Mapping Cardinality Constraints:

Mapping cardinality or cardinality ratio express the number of entities to which another entity can be associated via a relationship set. The mapping cardinality is most useful in describing binary relationship sets. (mapping cardinality also used for other relationship that is ternary etc.) For a binary relationship set the mapping cardinality must be one of the following types:

- One-to- one
- One-to- many
- Many-to-one
- Many-to-many

One-to-One:

An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. The following fig shows one to one mapping cardinality of entity sets A and B.

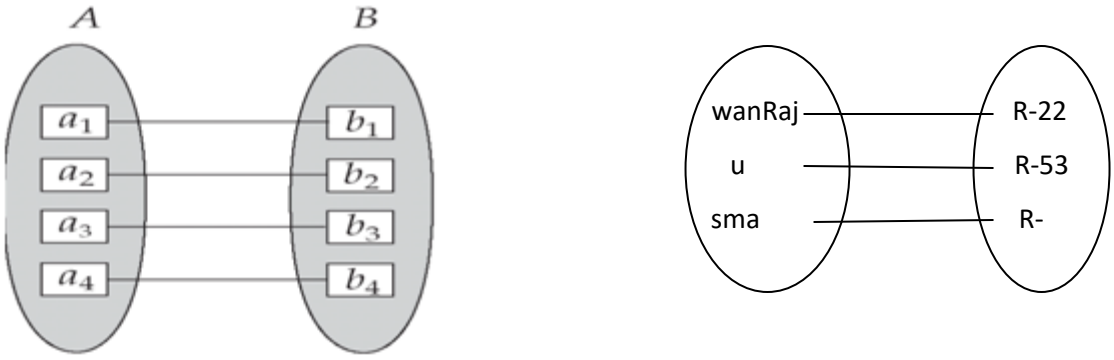


Fig: One-to-One

One-to-Many:

An entity in A is associated with any number (zero or more) of entities in B. An entity in B however can be associated with at most one entity in A. The following fig shows one-to-many mapping cardinality of entity sets A and B.

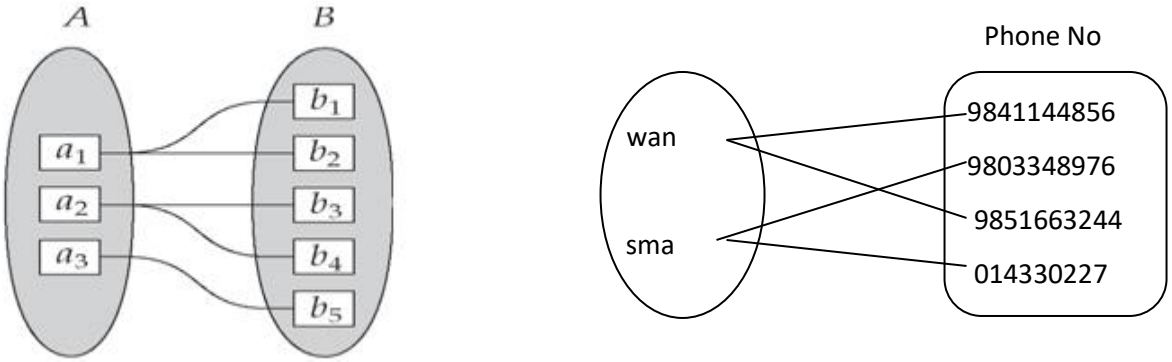


Fig: One-to-Many

Many-to-One:

An entity in A is associated with at most one entity in B. An entity in B, however, can be associated with any number (zero or more) of entities in A. The following fig clearly shows the many to one cardinality between entity sets A and B.

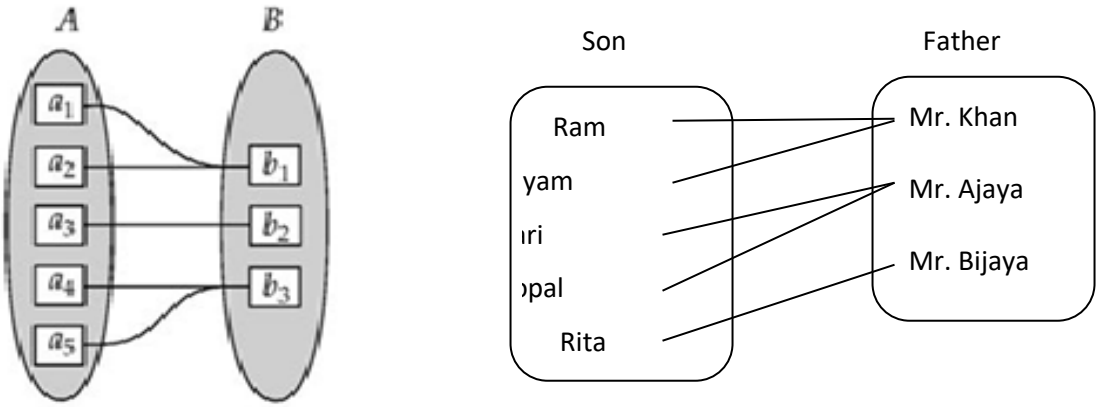


Fig: Many-to-One

Many-to-Many:

An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A. The following fig clearly shows the many-to-many cardinality between entity sets A and B.

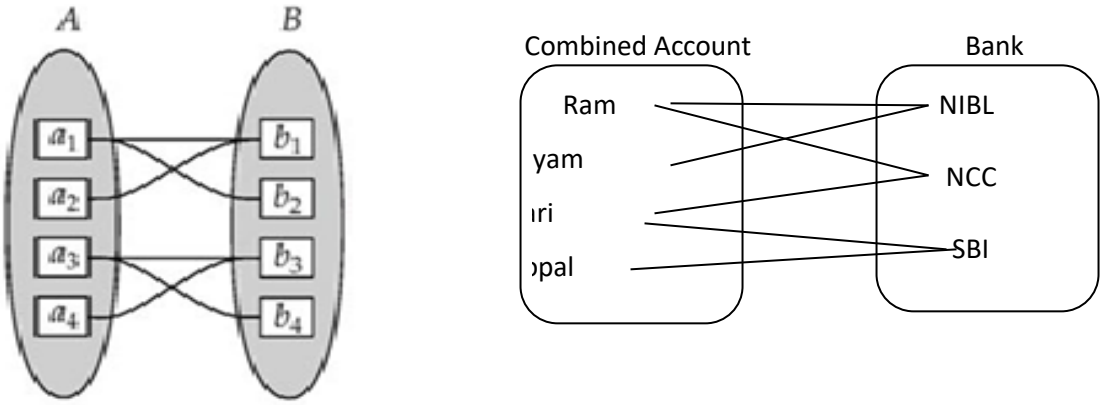


Fig: Many-to-Many

2. Participation Constraints:

The participation constraint specifies whether the existence of an entity depends on its being related to another entity via the relationship type.

There are two types of participation constraints:

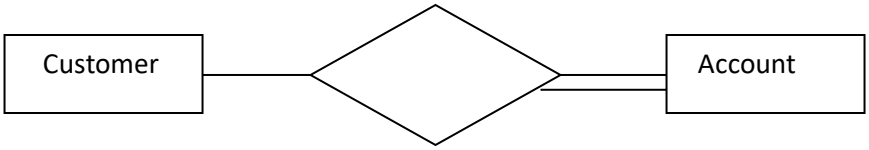
I. Total Participation Constraints

II. Partial Participation Constraints

Total Participation Constraints:

- If an entity can exist, only if it participates in at least one relationship instance, then that is called total participation, meaning that every entity in one set, must be related to at least one entity in a designated entity set.
- An example would be the Employee and Department relationship. If company policy states that every employee must work for a department, then an employee can exist only if it participates in at lest one relationship instance (i.e. an employee can't exist without a department)
- It is also sometimes called an existence dependency.
- Total participation is represented by a double line, going from the relationship to the dependent entity.

For example, consider customer and account entity sets in a banking system, and a relationship set depositor between them indicating that each customer must have an account. Then there is total participation of entity set account in the relationship set depositor.

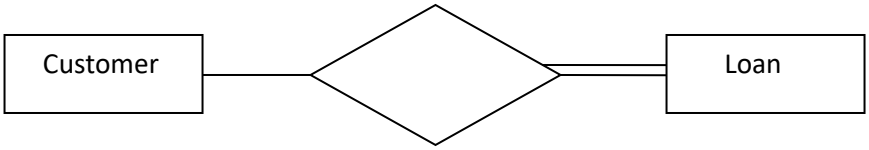


Note: Double lines are used to indicate that the participation of an entity set in a relationship set is total. And single lines are used to indicate that the participation of an entity set in a relationship set is partial.

Partial Participation Constraints:

- If only a part of the set of entities participate in a relationship, then it is called partial participation.
- Using the Company example, every employee will not be a manager of a department, so the participation of an employee in the "Manages" relationship is partial.
- Partial participation is represented by a single line.

. For example, consider customer and loan entity sets in a banking system, and a relationship set borrower between them indicating that some customers have loans. Then there is partial participation of entity set customer in the relationship set borrower.



Keys:-

KEY:-

Definition

A Column value in the table that uniquely identifies a single record in the table is called key of an table

A attribute or the set of attribute in the table that uniquely identifies each record in the entity set is called a key for that entity set

Types of keys

Simple Key: A key which has the single attribute is known as a simple key Composite key: A key which consist two or more attributes is called a Composite Key.

Example:

Cust_id is a key attribute of Customer Table it is possible to have a single key for one customer i.e is Cust_id ie Cust_id =1 is only for the Cust_name ="Yogesh" please refer to the Customer Table which is mentioned above.

Types of Key	Definition of Key
Super Key	A key is called super key which is sufficient to

	identify the unique record in the table
Candidate Key	A minimal super key is called candidate key. A super has no proper subset of candidate key.
Primary Key	A candidate key is chosen as a principal to identify a unique
Foreign Key	An column (or combination of columns) in the one table whose values is match the primary key in the another table

Types of key

1) Super Key

A key is called super key which is sufficient to identify the unique record in the table Customer Table

Cust_id	Cust_Name	Cust_Age	Cust_Address	Cust_Mobile_No	Cust_Phone_No
1	Yogesh	20	Kathmandu	9841567081	4434999
2	Ramesh	23	Pokhara	9985555522	4443434
3	Ram	18	Chitwan	9854333332	656565
4	Pramod	24	Dailekh	9848044401	34343434
5	Yashu	25	Birendranagar	9848080000	44545454
6	Sarawsati	23	Gulmi	9855555555	93434343

Example

Here Cust_id attribute of the entity set Customer is uniquely identify Customer entity from another so The Cust_id is the Super key. Another way is, the combination of Cust_id attribute and Cust_Name attribute is the Super key for the Customer Entity set. Only the Cust_Name is not called the Super Key because several customer may have the Same Name

2) candidate key

A minimal super key is called Candidate key .A super key has no proper subset of candidate key Here Minimum attribute of the super key is omitted unwanted attributed of an table that key is sufficient for identifying the unique record in the entity set so it is called as Candidate key The Candidate key is also known as the primary key

Cust_id	Cust_Name	Cust_Age	Cust_Address	Cust_Mobile_No	Cust_Phone_No
1	Yogesh	20	Kathmandu	9841567081	4434999
2	Ramesh	23	Pokhara	9985555522	4443434

3	Ram	18	Chitwan	9854333332	656565
4	Pramod	24	Dailekh	9848044401	34343434
5	Yashu	25	Birendranag ar	9848080000	44545454
6	Sarawsati	23	Gulmi	9855555555	93434343

From above statement say combination of Cust_id attribute and Cust_Name is a super key for the Customer entity set it is required to distinguish one record on the Customer entity from another record of sane set. But Cust_id attribute of the Customer entity is aslo known as minimal super key which also enough to distinguish one record from customer entity from another record from customer entity set,
because Cust_Name is th additional attribute of the Csutomer table

3) Primary key

Primary key of the table is a columns or combination of the some columns whose values is uniquely identify a single record in the table. Primary key state no two record of the table contain the same value in that column or Cobination of the column It state that a unique identifier for the entity set.

Cust_id	Cust_Name	Cust_Age	Cust_Address	Cust_Mobile_No	Cust_Phone_No
1	Yogesh	20	Kathmandu	9841567081	4434999
2	Ramesh	23	Pokhara	9985555522	4443434
3	Ram	18	Chitwan	9854333332	656565
4	Pramod	24	Dailekh	9848044401	34343434
5	Yashu	25	Birendranag ar	9848080000	44545454
6	Sarawsati	23	Gulmi	9855555555	93434343

hence the customer age column contain repeated values and Customer Name also cannot act as primary key because it earlier state that several customer may have the same name hence Cust_Name column has the repeated values .Hence there Cust_id can act as the primary key in the Customer table this is only column which contain a unique set of values.

Foreign Key

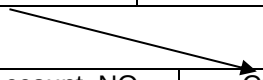
A Column (or combination of Columns) in the one tables whose values is match the primary key in the another table is called as a foreign key Foreign key can also have one or more column like as primary key A single table may contain more than one foreign key which is related to the more tah one table, the table which used the foreign key is said the referential integrity

What the referential integrity

Referential integrity say the column which contain foreign key in one table must be primary key of another table In general term, Foreign key of Table A must be Primary key of Table B

Example
Customer Table

Cust_id	Cust_Name	Cust_age	Cust_address	Cust_mobile_no	Cust_phone_no



Account_NO	Cust_Id	Account_Type	Balance	Description

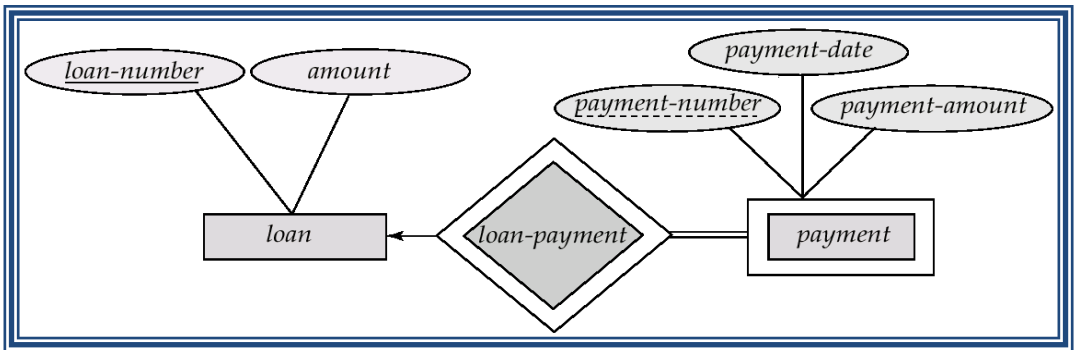
In The above example Cust_id is the primary key for the customer Table while Cust_id is the foreing key for the Account table. Here the the datatype assigned to column and Numder of column in the foreign key is same as to the primary key.

Cust_id	Cust_Name	Cust_Age	Cust_Address	Cust_Mobile_No	Cust_Phone_No
1	Yogesh	20	Kathmandu	9841567081	4434999
2	Ramesh	23	Pokhara	9985555522	4443434
3	Ram	18	Chitwan	9854333332	656565
4	Pramod	24	Dailekh	9848044401	34343434
5	Yashu	25	Birendranagar	9848080000	44545454
6	Sarawsati	23	Gulmi	9855555555	93434343

Account_NO	Cust_id	Account_type	Balance	Description
101	1	Saving	10,000	
102	2	Saving	20,000	
103	2	Saving	20,200	
104	3	Current	11,000	
105	4	Saving	50,000	

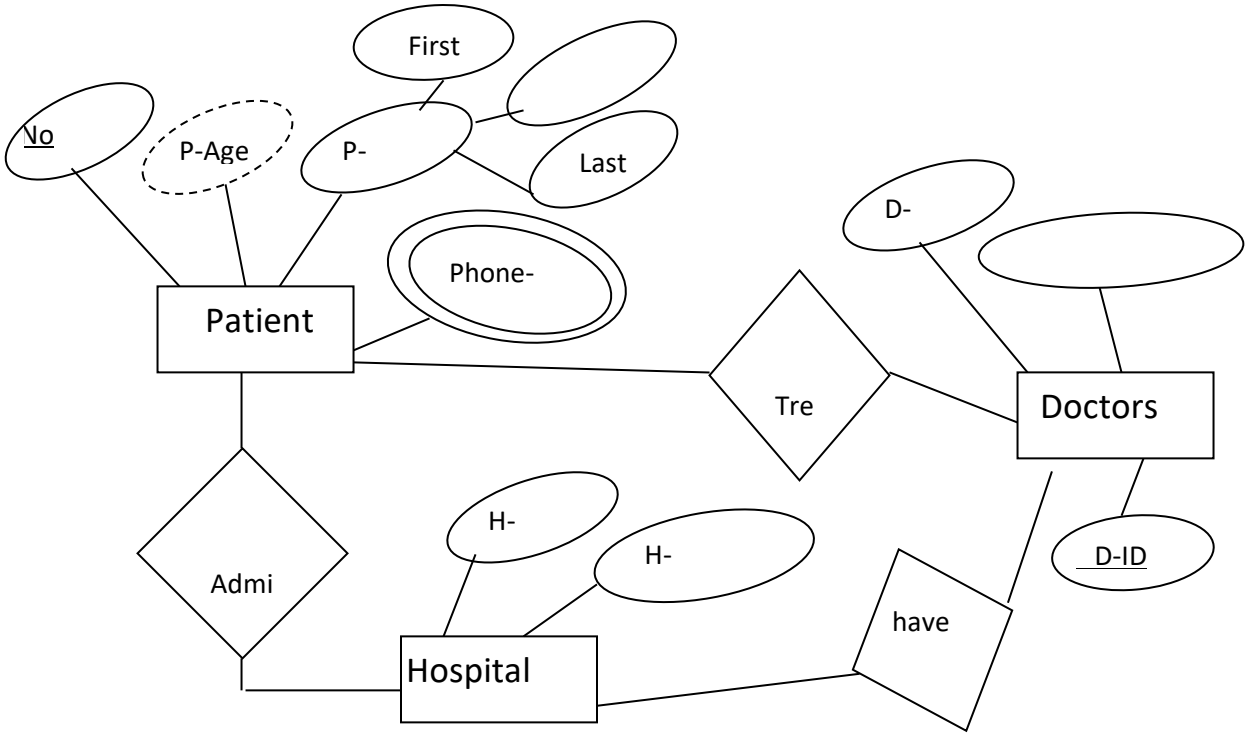
Weak entity set:

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed as a weak entity set. An entity set that has a primary key is termed as a strong entity set. For a weak entity set to be meaningful, it must be associated with another entity set, called the identifying or owner entity set, using one of the key attribute of owner entity set. The relationship associating the weak entity set with the identifying entity set is called the identifying relationship. An attribute of weak entity set that is use in combination with primary key of the strong entity set to identify the weak entity set uniquely is called discriminator (partial key).



In the above figure, *payment-number* is partial key and (*loan-number, payment-number*) is primary key for *payment* entity set.

Example: E-R diagram for hospital with a set of patients and medical doctors.



Extended E-R model (EER model):

The EER model includes all of the concepts introduced by the ER model. Additionally it includes the concepts of a subclass and superclass, along with the concepts of specialization and generalization.

There are basically four concepts of EER-Model:

- Attribute Inheritance (subclass / superclass relationship)
- Specialization

- Generalization
- Categories
- Aggregation

Subclass and superclass:

The class that is derived from another class is called a ***subclass***. The class from which a subclass derives is called the***superclass***. The following figure illustrates these two types of classes:

- ✓ An entity type may have additional meaningful sub-groupings of its entities. Example: EMPLOYEE may be further grouped into {SECRETARY, ENGINEER, TECHNICIAN, MANAGER, MANAGER, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE …}
- ✓ EER diagrams extend ER diagrams to represent these additional sub-groupings, called ***subclasses* or *subtypes.*** Each of these subgroups is called a **subclass** of the EMPLOYEE entity type.
- ✓ The EMPLOYEE entity type is called the ***superclass*** of each of these subclasses.
- ✓ The relationship between a ***superclass*** and any one of its subclasses is called a **superclass/subclass** or **class/subclass** or **IS-A (IS-AN) relationship** (e.g. EMPLOYEE/ SECRETARY EMPLOYEE/MANAGER).
- ✓ Subclass entities have their own specific attributes. They alsoinheritall attributes and relationships of its ***superclass*** (subclasses can be considered as separate entity types).

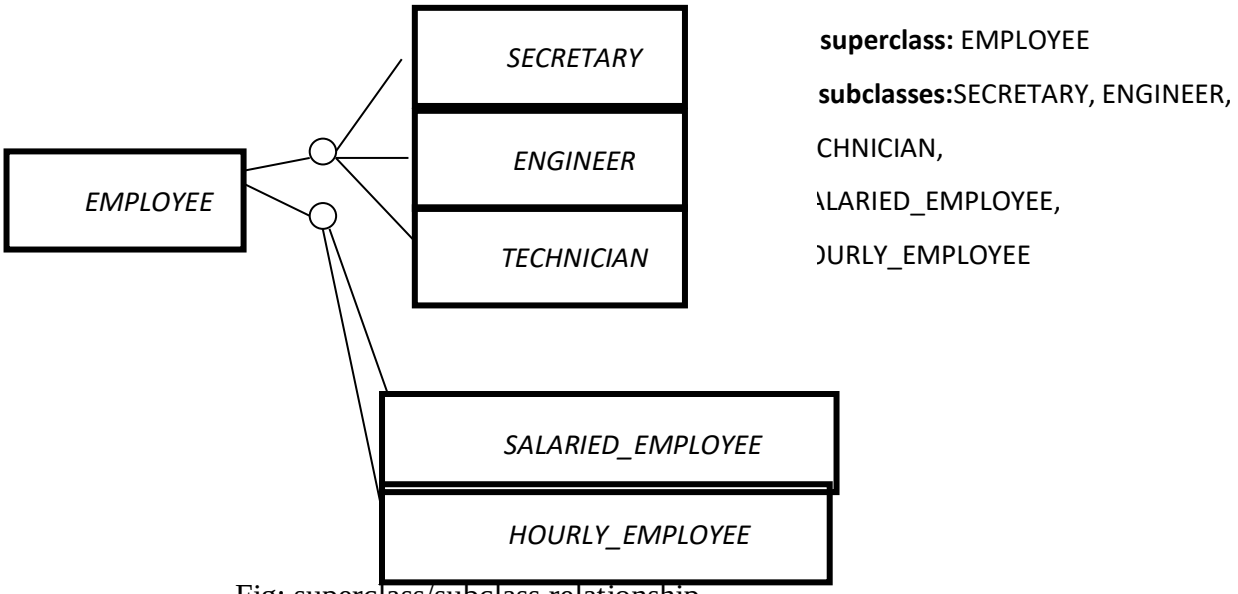


Fig: superclass/subclass relationship

Specialization:

The process of defining a set of subclasses from a superclass is known as specialization.

The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass. It is a **top-down design** process.

Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type*.

Note: There may have several specializations of the same superclass.

Generalization:

It is a **bottom-up design** process. Here, we combine a number of entity sets that share the same features into a higher-level entity set. The original classes become the subclass of the newly formed generalized superclass. The reason, a designer applies generalization is to emphasize the similarities among the entity sets and hide their differences. Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way. The terms specialization and generalization are used interchangeably.

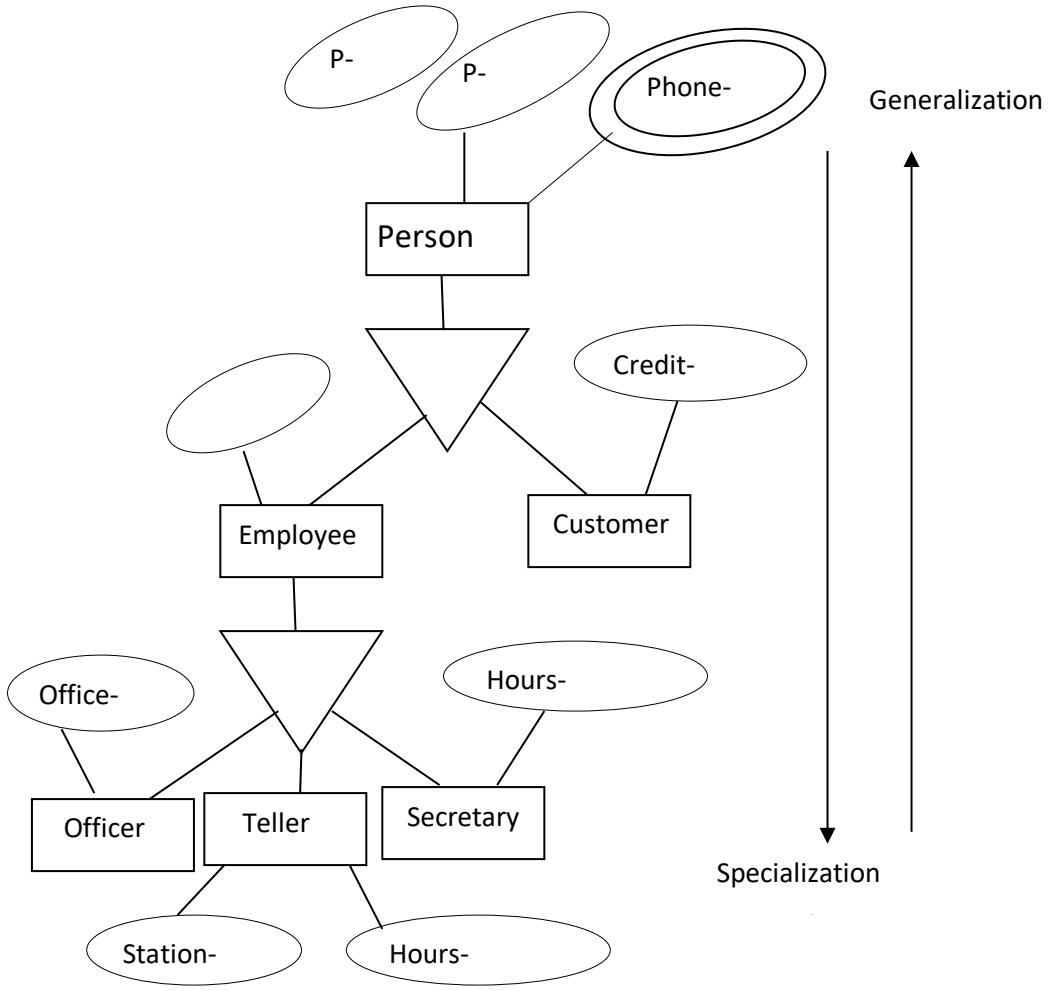


Fig: ER diagram with specialization and generalization

Note:

- In EER disjoint-constraint is illustrated by placing the letter **d** inside the circle
- In case **overlap** between subclasses is allowed, we place the letter **o** inside the circle.

Constraints on Specialization and Generalization:

Two basic constraints can apply to a specialization/generalization:

- **Disjointness Constraint:**
- **Completeness Constraint:**

Disjointness/Overlapping Constraints:

It specifies that the subclasses of the specialization must be ***disjoint***. Here an entity can be a member of at most one of the subclasses of the specialization and it is represented by ***d*** in EER diagram.
If not disjoint, specialization is ***overlapping*.** That is the same entity may be a member of more than one subclass of the specialization and it is represented by ***o*** in EER diagram.

Completeness Constraint:

Total participation constraint specifies that every entity in the superclass must be a member of some subclass in the specialization/generalization. It is represented by ***double line*** in EER diagram.

Partial participation constraint allows an entity not to belong to any of the subclasses and shown in EER diagrams by a ***single line***.

Aggregation:

One of the limitations of E-R model is that it cannot express relationship among relationships. One of the solutions in such a situation is using aggregation. Aggregation is an abstraction in which relationship sets (along with their associated entity sets) are treated as higher-level entity sets and can participate in relationships.

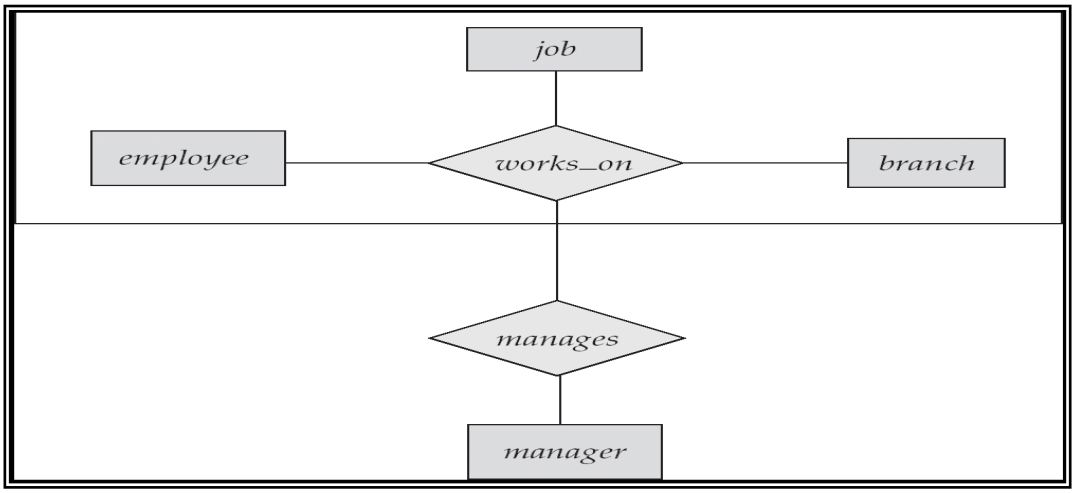


Fig: E-R Diagram with Aggregation

Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship *works-on*, which we saw earlier, between a *employee*, *branch*, and *job* see below figure. Now, suppose we want to record managers for tasks performed by an employee at a branch; that is, we want to record managers for (*employee*, *branch*, *job*) combinations. Let us assume that there is an entity set *manager*.

One alternative for representing this relationship is to create a quaternary relationship *manages* between *employee*, *branch*, *job*, and*manager*. (A quaternary relationship is required—a binary relationship between *manager* and *employee* would not permit us to represent which (*branch*, *job*) combinations of an employee are managed by which manager.) Using the basic E-R modeling constructs, we obtain the E-R diagram of Figure 2.18. (We have omitted the attributes of the entity sets, for simplicity.) It appears that the relationship sets *works-on* and *manages* can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some *employee*, *branch*, *job* combinations many not have amanager.

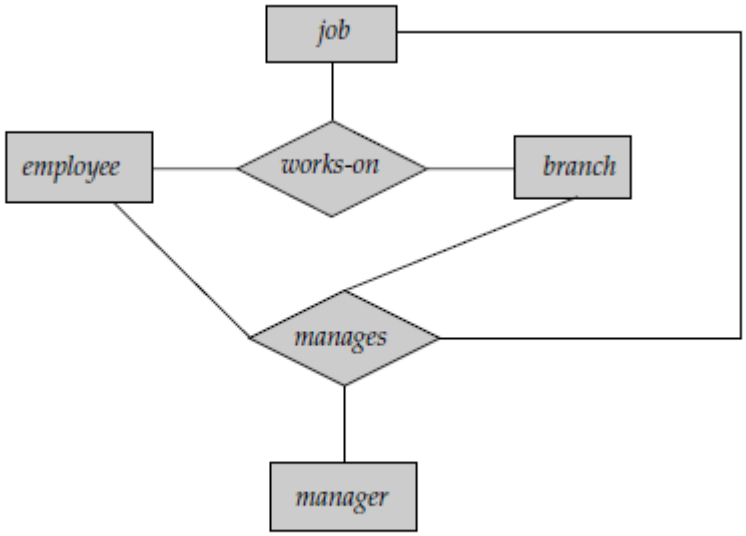


Figure 2.18 E-R diagram with redundant relationships.

There is redundant information in the resultant figure, however, since every *employee*, *branch*, *job* combination in *manages* is also in *works-on*. If the manager were a value rather than an *manager* entity,we could instead make *manager*amultivalued attribute of the relationship *works-on*. But doing so makes it more difficult (logically as well as in execution cost) to find, for example, employee-branch-job triples for which a manager is responsible. Since the manager is a *manager* entity, this alternative is ruled out in any case.

The best way to model a situation such as the one just described is to use aggregation. **Aggregation** is an abstraction through which relationships are treated as higherlevel entities. Thus, for our example, we regard the relationship set *works-on* (relating

the entity sets *employee*, *branch*, and *job*) as a higher-level entity set called *works-on*. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship *manages* between *works-on* and *manager* to represent who manages what tasks.

Reduction of E-R Schema to tables:

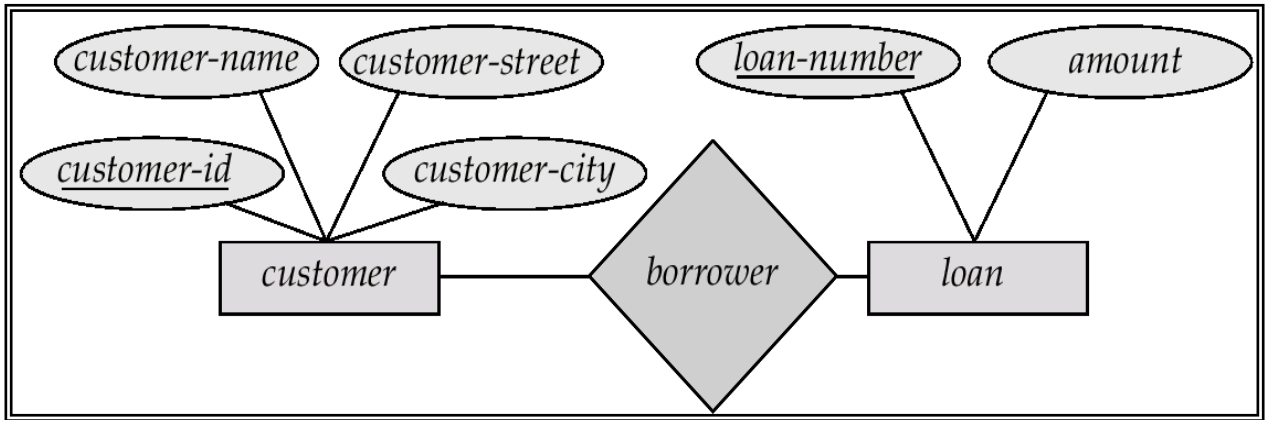
To reduce given ER diagram into table simply we create a table for each entity set and for each relationship sets. And that assigned the name of the corresponding entity set or relationship set as table name. Generally the number of attributes of an entity set or relationship set equal to the degree of a corresponding table (fields of a table).

To reduce given ER diagram into tables normally we divide ER diagram into three sections:

- Strong entity sets
- Weak entity sets and
- Relation sets
-

Reducing strong entity sets into tables:

Consider an E-R diagram as given below



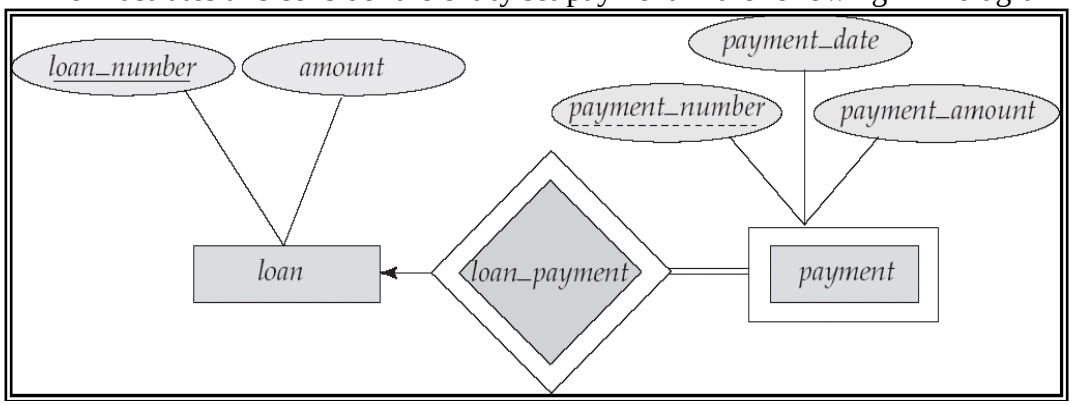
The tabular representation of the entity set loan of the given E-R diagram, This entity has two attributes loan-number and amount. We represent this entity set by a table called loan, with two columns named loan-number and amount as below:

Loan

loan_number	amount
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Reducing weak entity sets into tables:

To illustrates this consider the entity set payment in the following E-R diagram



The table of entity'*payment*' consist the column names ***loan-number, payment-number, payment-data, and payment-amount as below:***

Payment

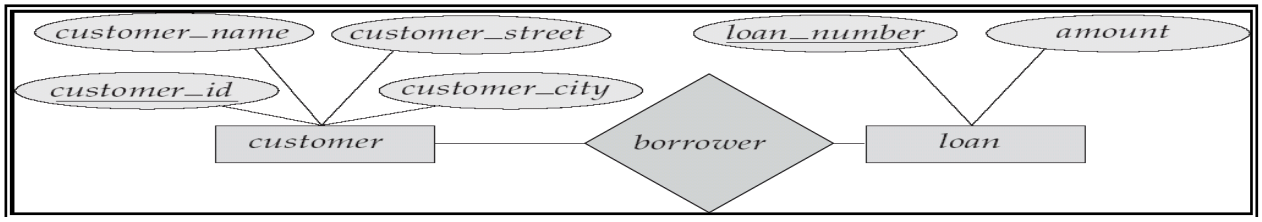
Loan-No	***Payment-No***	***Payment-Date***	***Payment-amount***
L-11	L-11	2069-02-22	50,000
L-22	L-22	2069-04-28	70,000
L-07	L-07	2069-01-19	45,000
L-32	L-32	2070-02-02	98,000

Reducing Relationship sets into tables:

To explain this, consider a relationship set ***borrower*** in E-R diagram and this relationship set involves the following entity sets:

- ***Customer*** with the primary key *customer-id*
- ***Loan*** with the primary key l*oan-number*.

This relationship set does not have any its own descriptive attributes, so the ***borrower*** table has two columns labeled as *customer-id* and *loan-number*.



Borrower

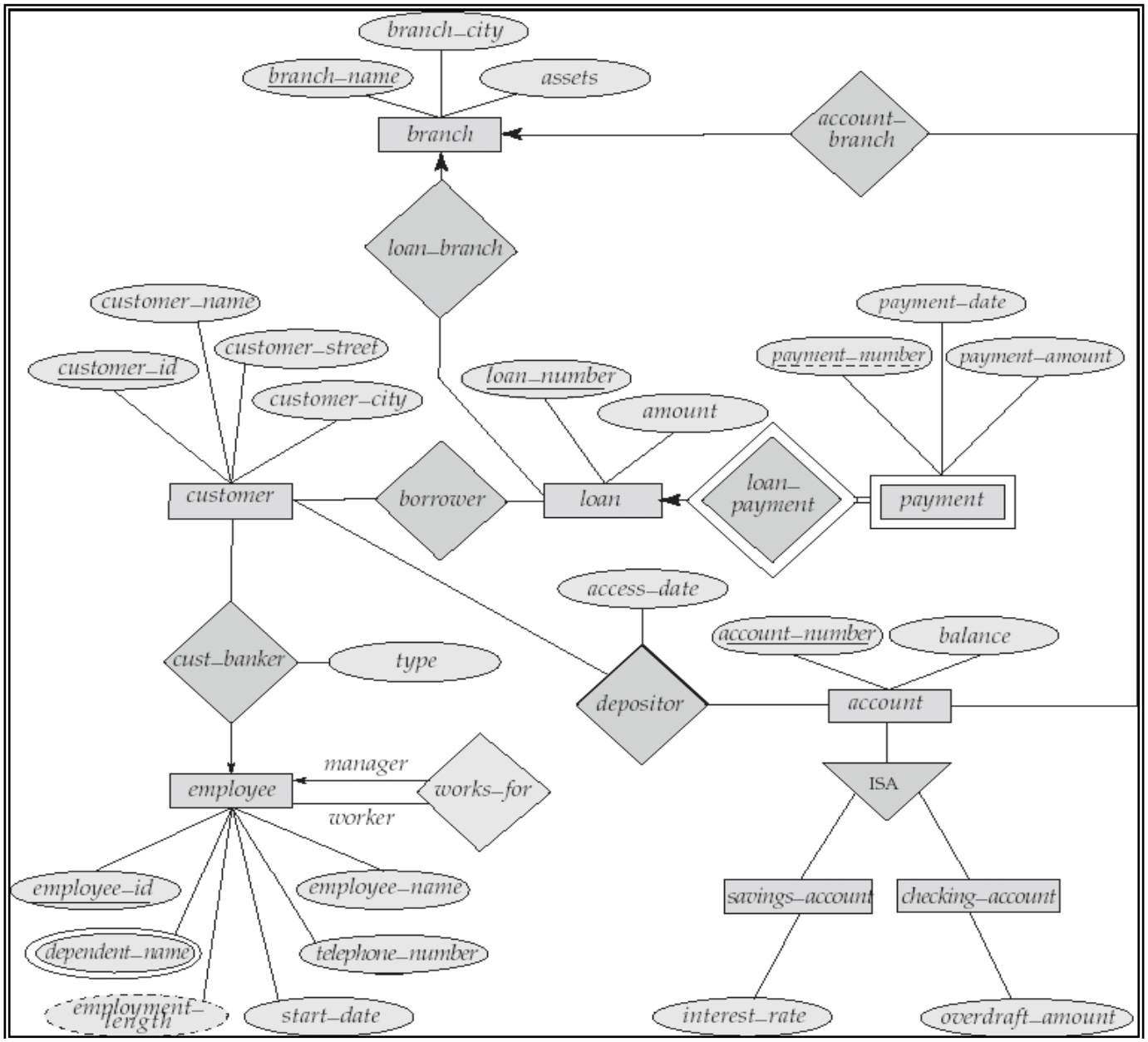
Customer-id	***Loan-number***
019-29-3746	L-11
019-28-3123	L-17
019-24-3144	L-76

E-R Diagram for a Banking Enterprise

The E-R diagram for the banking enterprise is given below.

A bank has many branches in a country. Each branch is identified with the help of branch name. Each branch of the bank has multiple customers. Customers also include employees of that particular branch. An employee can be a manager or staff of that branch. Branch gives each customer an id in order to identify them. Customers can open two types of accounts i.e. saving account and Checking account. Branches of the bank provides Loan to the customers if required. Re-Payment of loan is done with the help of payment number of that particular loan.

Draw an extended ER diagram with the help of above details.



Example: ER diagram for Company database system

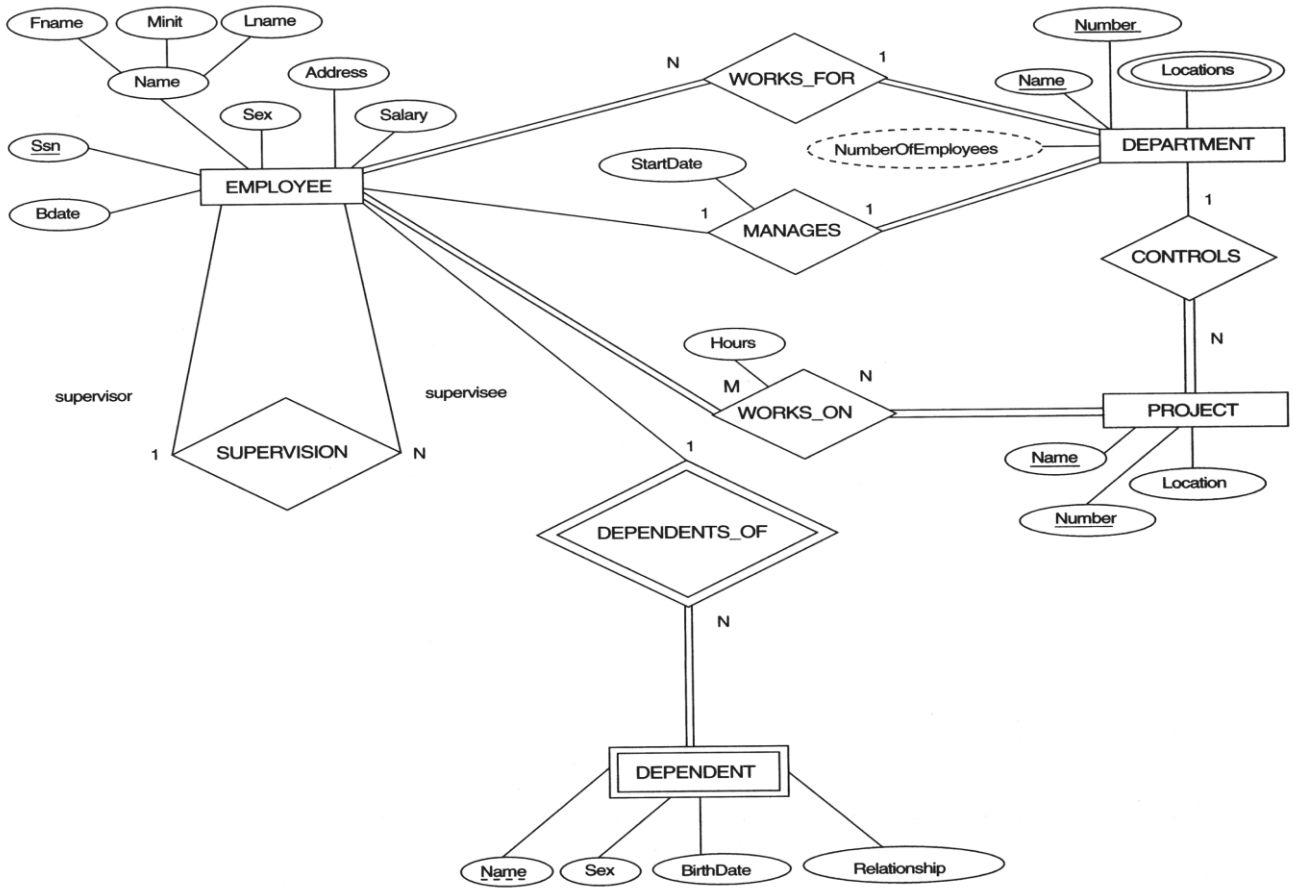
sample database application, called COMPANY, which serves to illustrate the basic ER model concepts and their use in schema design. The COMPANY database keeps track of a company's employees, departments, and projects. Suppose that after the requirements collection and analysis phase, the database designers provide the following description of the mini-world—the part of the company that will be represented in the database.

■ The company is organized into departments. Each department has a unique name, a unique number, and a particular employee who manages the department. We keep track of the start date when that employee began managing the department. A department may have several locations.

■ A department controls a number of projects, each of which has a unique name, a unique number, and a single location.

■ We store each employee's name, Social Security number, address, salary, sex (gender), and birth date. An employee is assigned to one department, but may work on several projects, which are not necessarily controlled by the same department. We keep track of the current number of hours per week that an employee works on each project. We also keep track of the direct supervisor of each employee (who is another employee).

■ We want to keep track of the dependents of each employee for insurance purposes. We keep each dependent's first name, sex, birth date, and relationship to the employee.



Example 3: Construct an E-R diagram for a hospital with a set of patients and a set of medicaldoctors. Associate with each patient a log of the various tests and examinationsconducted.

Or

patients (patient-id, name, insurance, date-admitted, date-checked-out)
doctors (doctor-id, name, specialization)
test (testid, testname, date, time, result)
doctor-patient (patient-id, doctor-id)
test-log (testid, patient-id) performed-by (testid, doctor-id)

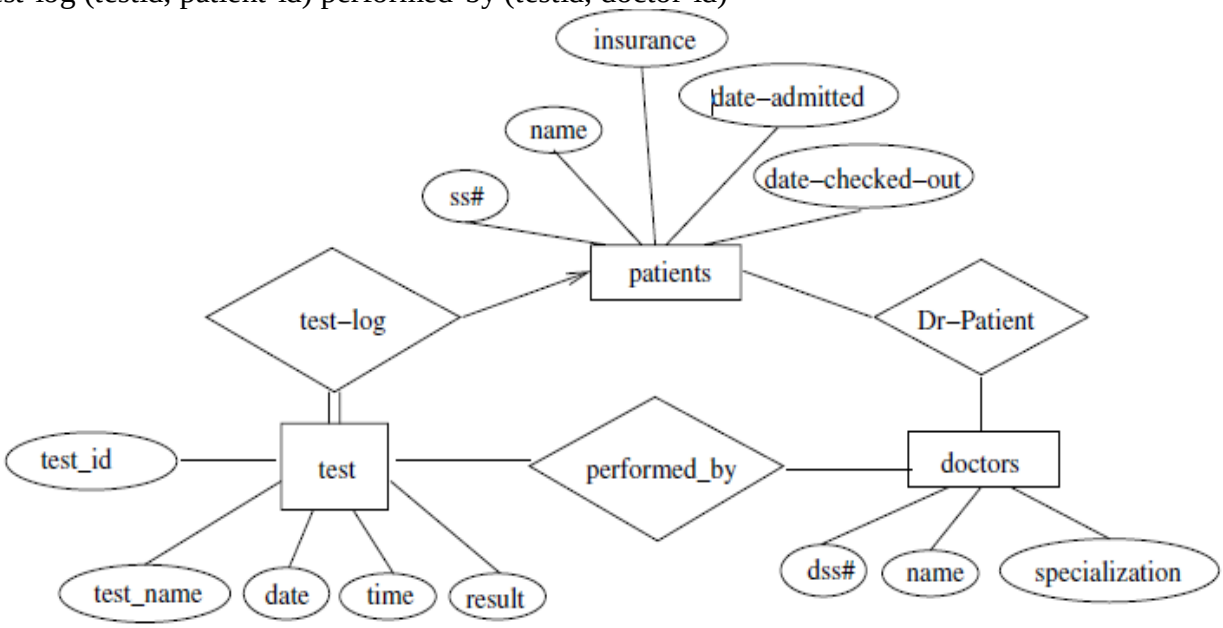


Figure E-R diagram for a hospital.

Example 4 Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

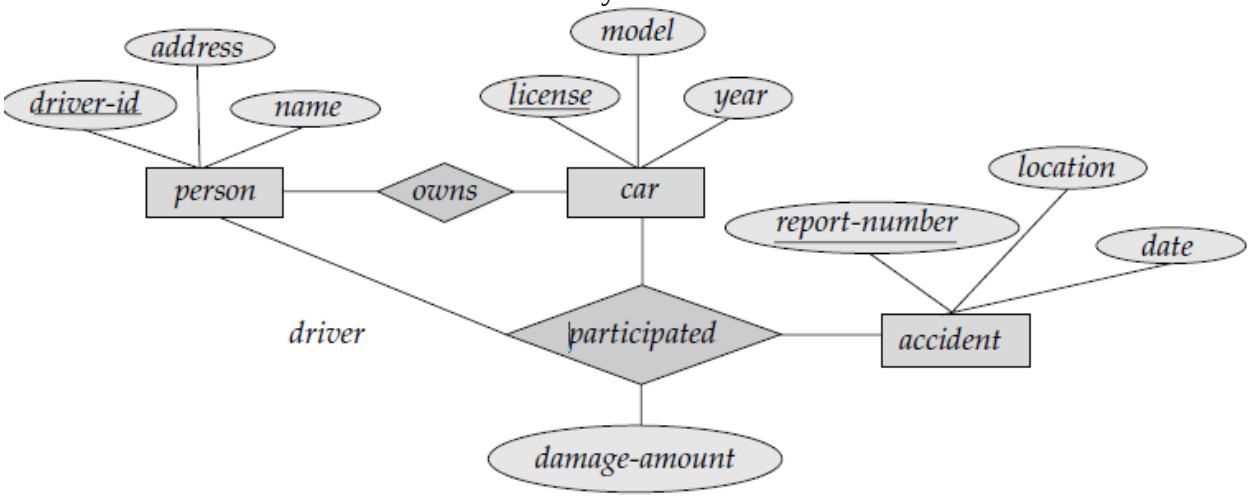
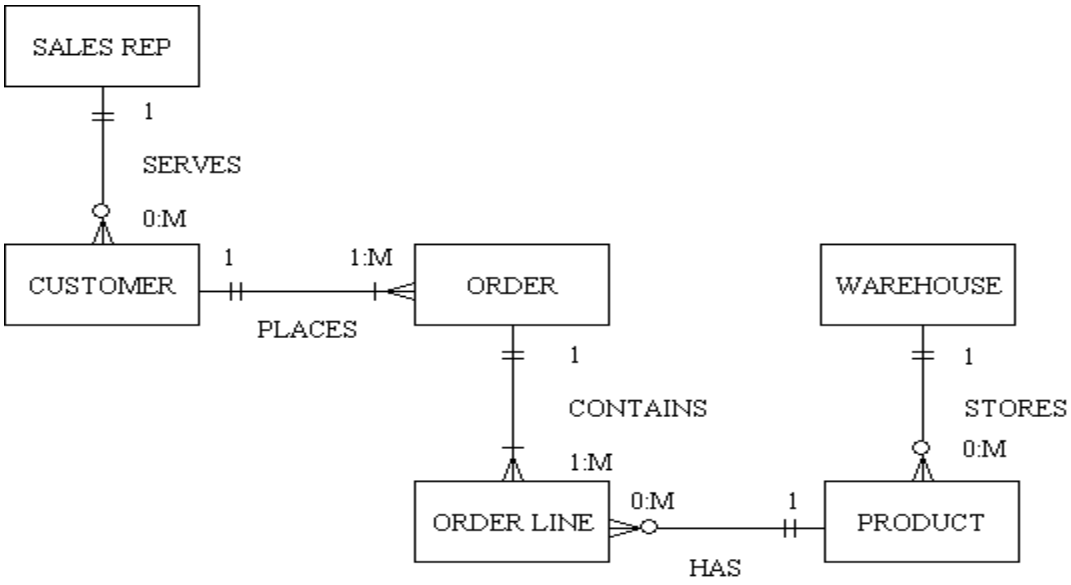


Figure E-R diagram for a Car-insurance company.

Figure 8 - ERD with ORDER and ORDER_LINE



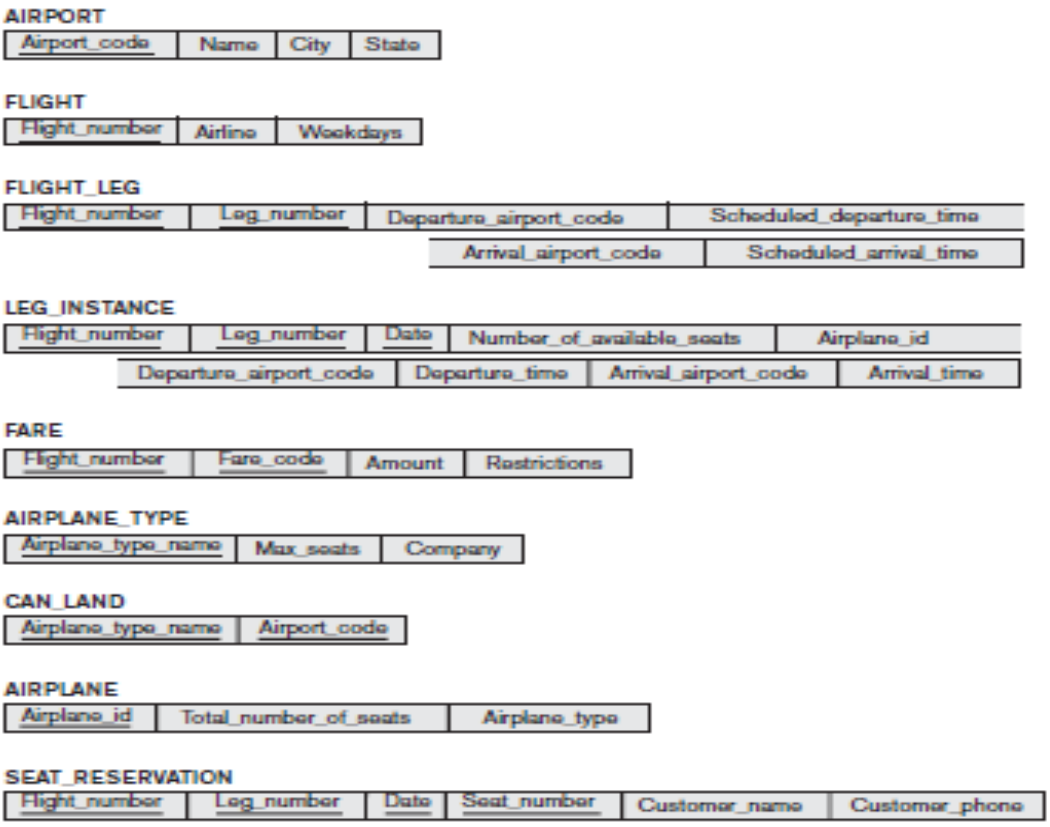


Figure 3.8
The AIRLINE relational database schema.

3.14. Consider the following six relations for an order-processing database application in a company:

CUSTOMER(Cust#, Cname, City)
ORDER(Order#, Odate, Cust#, Ord_amt)
ORDER_ITEM(Order#, Item#, Qty)

ITEM(Item#, Unit_price)
SHIPMENT(Order#, Warehouse#, Ship_date)
WAREHOUSE(Warehouse#, City)

Here, Ord_amt refers to total dollar amount of an order; Odate is the date the order was placed; and Ship_date is the date an order (or part of an order) is shipped from the warehouse. Assume that an order can be shipped from several warehouses. Specify the foreign keys for this schema, stating any assumptions you make. What other constraints can you think of for this database?

3.15. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:

SALESPERSON(Ssn, Name, Start_year, Dept_no)
TRIP(Ssn, From_city, To_city, Departure_date, Return_date, Trip_id)
EXPENSE(Trip_id, Account#, Amount)

A trip can be charged to one or more accounts. Specify the foreign keys for this schema, stating any assumptions you make.

3.16. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(Ssn, Name, Major, Bdate)
COURSE(Course#, Cname, Dept)
ENROLL(Ssn, Course#, Quarter, Grade)
BOOK_ADOPTION(Course#, Quarter, Book_isbn)
TEXT(Book_isbn, Book_title, Publisher, Author)

Specify the foreign keys for this schema, stating any assumptions you make.

3.17. Consider the following relations for a database that keeps track of automobile sales in a car dealership (OPTION refers to some optional equipment installed on an automobile):

CAR(Serial_no, Model, Manufacturer, Price)
OPTION(Serial_no, Option_name, Price)
SALE(Salesperson_id, Serial_no, Date, Sale_price)
SALESPERSON(Salesperson_id, Name, Phone)

First, specify the foreign keys for this schema, stating any assumptions you make. Next, populate the relations with a few sample tuples, and then give an example of an insertion in the SALE and SALESPERSON relations that *violates* the referential integrity constraints and of another insertion that does not.

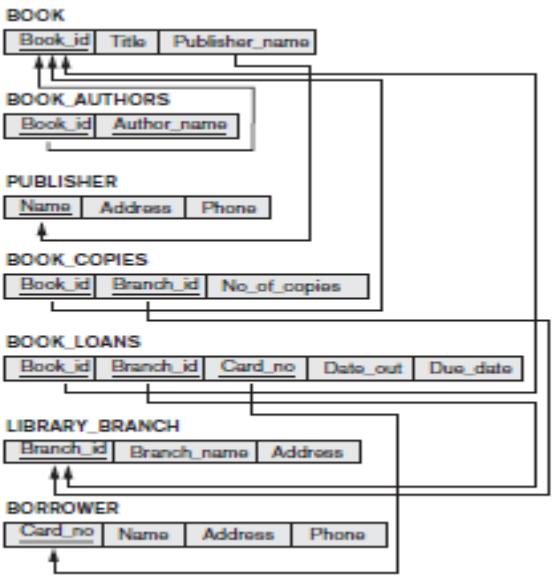
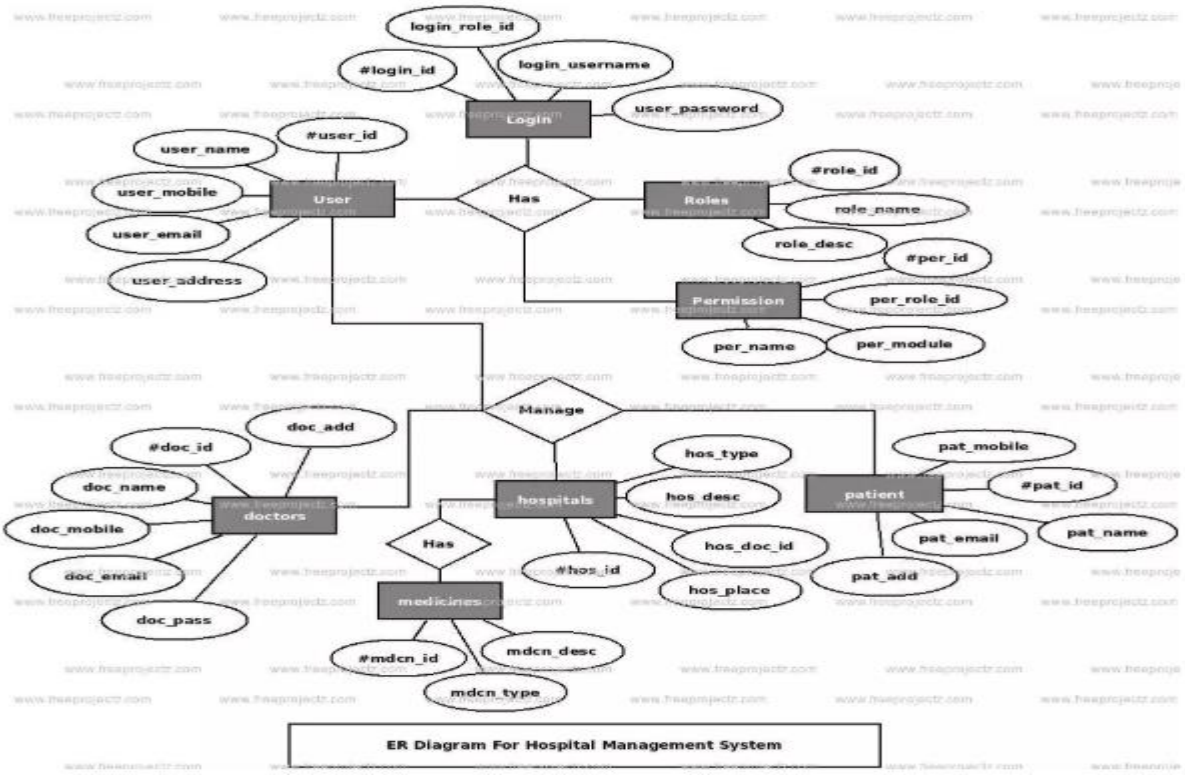


Figure 4.6
A relational database schema for a LIBRARY database.

Hospital Management System entities and their attributes :

- **Hospitals Entity** : Attributes of Hospitals are hospital_id, hospital_doctor_id, hospital_name, hospital_place, hospital_type, hospital_description, hospital_address
- **Patient Entity** : Attributes of Patient are patient_id, patient_name, patient_mobile, patient_email, patient_username, patient_password, patient_address, patient_blood_group
- **Doctors Entity** : Attributes of Doctors are doctor_id, doctor_name, doctor_specialist, doctor_mobile, doctor_email, doctor_username, doctor_password, doctor_address
- **Nurses Entity** : Attributes of Nurses are nurse_id, nurse_name, nurse_duty_hour, nurse_mobile, nurse_email, nurse_username, nurse_password, nurse_address,
- **Appointments Entity** : Attributes of Appointments are appointment_id, appointment_doctor_id, appointment_number, appointment_type, appointment_date, appointment_description
- **Medicines Entity** : Attributes of Medicines are medicine_id, medicine_name, medicine_company, medicine_composition, medicine_cost, medicine_type, medicine_dose, medicine_description



ER Diagram For Hospital Management System

- A university registrar's office maintains data about the following entities: (a) courses, including course number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Furthermore, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.

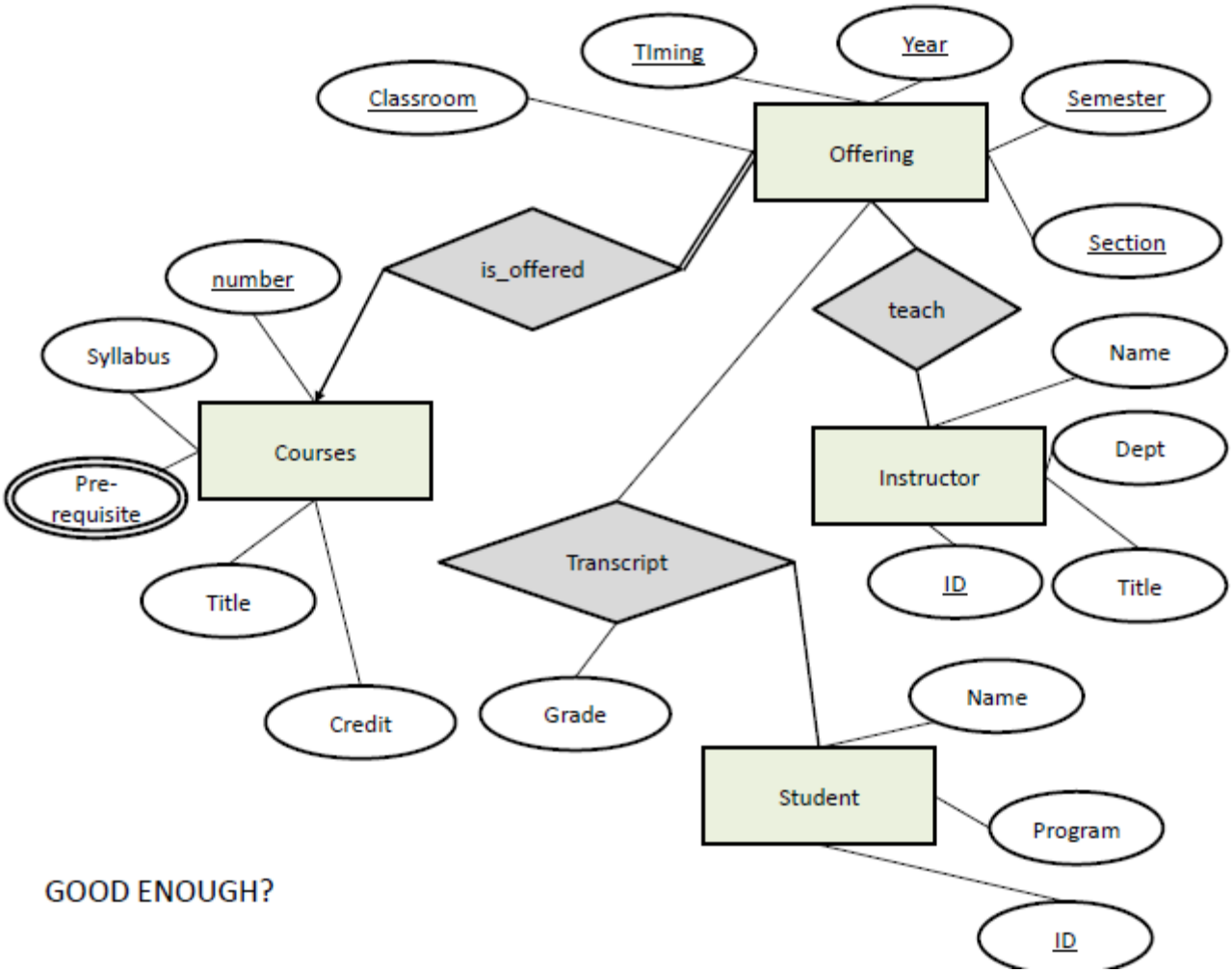
Construct an E-R diagram for the registrar's office.

Document all assumptions that you make about the mapping constraints.

- A university registrar's office maintains data about the following entities: (a) courses, including course number, title, credits, syllabus, and prerequisites; (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom; (c) students, including student-id, name, and program; and (d) instructors, including identification number, name, department, and title. Furthermore, *the enrollment of students in courses and grades* (transcript) *awarded to students in each course they are enrolled for must be appropriately modeled.*

Construct an E-R diagram for the registrar's office.

Document all assumptions that you make about the mapping constraints.



GOOD ENOUGH?

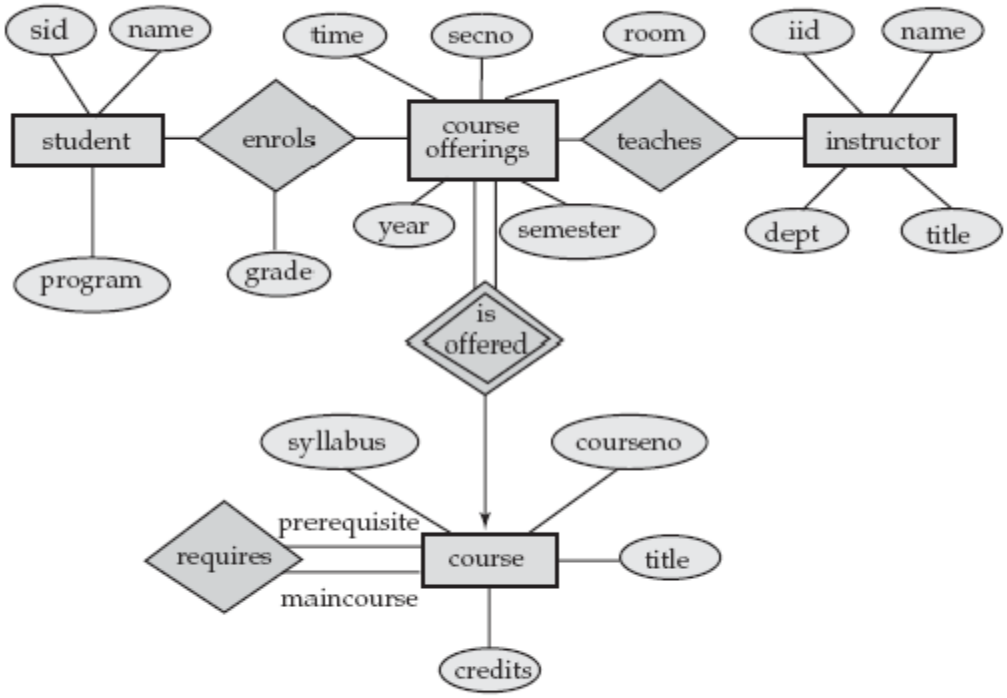


Figure 6.2 E-R diagram for a university.

INSURANCE COMPANY

Design a database for insurance company.Asume that following are the requirements that were collected:

An insurance company has different policies.Policies have pno,term_price and coverage.

Policies are categorized based on their types.There are two types:Auto_policy and Home_policy.

Policies for vehicles come under Auto policy.Auto_policy has pno,vehicle type and issue date.

Policies for house come under home policy.Home_policy has pno,issue date and term_price.

Customers take policies policy through policy agent.A customer can take only one policy .

ANSWER

ENTITIES:

1)Policy

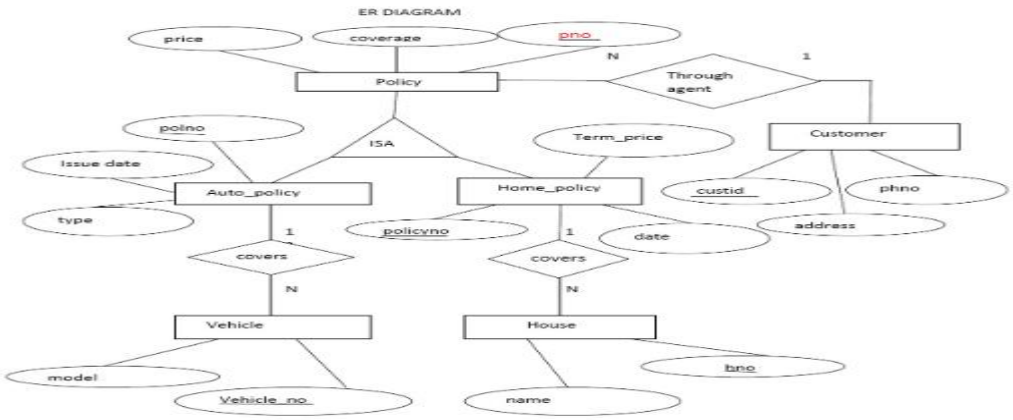
2)Auto_policy

3)Home_policy

4)Vehicle

5)House

6)Customer



Create an E-R diagram of a sports club conducted by school.

a) A school is decided to setup a sports club outside the school.

b) Sports club can be categorized based on the type: cricket club, football club.

c) A student can join in any one of the sports club.

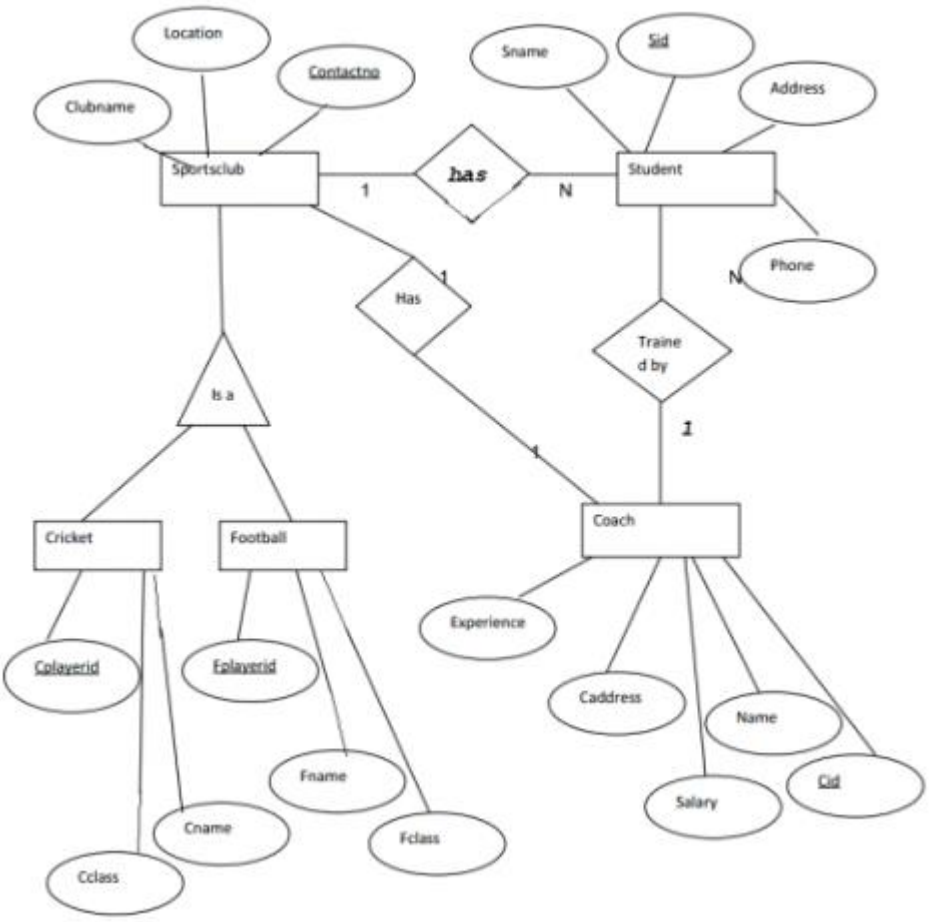
d) Each sports club has a coach who trains the students.

e) Each student can be identified using id no.

f) Salary, experience, name of the coach can also be included.

ANSWER:

STEP 5: CREATE E-R DIAGRAM



Unit 4

Structured Query Language (SQL)

- Basic structure
- Set operation
- Aggregate functions
- NULL values
- Nested sub queries
- Views
- Modification of database
- joined relations
- Data definition languages (DDL)
- Other SQL features:

  Dynamic and Embedded SQL

Introduction:

SQL is acomputer language for organizing, managing, and retrieving data stored by a computer database. In fact, SQL works with one specific type of database, called ***a relational database***.The name "SQL" is the short form for ***Structured Query Language***.

SQL is used to control all of the functions that a DBMS provides for its users, including:

1. **Data definition:** SQL lets a user define the structure and organization of the stored data and relationships among the stored data items.
2. **Data retrieval:** SQL allows a user or an application program to retrieve stored data from the database and use it.
3. **Data manipulation:** SQL allows a user or an application program to update the database by adding new data, removing old data, and modifying previously stored data.
4. **Access control:** SQL can be used to restrict a user's ability to retrieve, add, and modify data, protecting stored data against unauthorized access.
5. **Data sharing:** SQL is used to coordinate data sharing by concurrent users, ensuring that they do not interfere with one another.
6. **Data integrity:** SQL defines integrity constraints in the database, protecting it from corruption due to inconsistent updates or system failures.

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database

SQL DML and DDL:

SQL can be divided into two parts: The Data Manipulation Language (DML) and the Data Definition Language (DDL).

The query and update commands form the DML part of SQL:

- **SELECT:** - extracts data from a database
- **UPDATE:** - updates data in a database
- **DELETE:**- deletes data from a database
- **INSERT INTO:** - inserts new data into a database

The DDL part of SQL permits database tables to be created or deleted. It also defines indexes (keys), specify links between tables, and impose constraints between tables.
The most important DDL statements in SQLare:

- **CREATE DATABASE**- creates a new database
- **ALTER DATABASE**- modifies a database
- **CREATE TABLE**- creates a new table
- **ALTER TABLE**- modifies a table
- **DROP TABLE**- deletes a table
- **CREATE INDEX**- creates an index (search key)
- **DROP INDEX**- deletes an index

Basic structure:

The basic structure of an SQL expression consists of three clauses: **SELECT, FROM** and **WHERE.**

- The SELECT clause corresponds to the projection operation of relational algebra. It is used to list the attributes desired in the result of a query.
- The FROM clause corresponds to the Cartesian product operation of relational algebra. It is used to list the relations to be used in the evaluation of the expression.
- The WHERE clause corresponds to the selection predicate of relational algebra. It consists of a predicate in the attributes of the relations that appear in the FROM clause.

A typical SQL query has the form

SELECT $A_1, A_2, \ldots\ldots, A_n$
FROM $R_1, R_2, \ldots\ldots, R_n$
WHERE P

Where each $A_i$represents an attribute, each R_i is a relation and P is a predicate.
Its equivalent relational algebra expression is:

$$\Pi_{A1,\, A2,\, \ldots\ldots,An}(\sigma_p(R_1 X\, R_2 X\, \ldots\ldots X\, R_n))$$

The SQL SELECT Statement:

The SELECT statement is used to select data from a database. The result is stored in a result table, called theresult-set.

SQL SELECT Syntax:

SELECT column_name(s)FROM table_name
and
SELECT *FROMtable_name

Note: SQL is not case sensitive. SELECT is the same as select.

Sailors(*sid:* integer, *sname:* string, *rating:* integer, *age:* real)
Boats(*bid:* integer, *bname:* string, *color:* string)
Reserves(*sid:* integer, *bid:* integer, *day:* date

Sailors BoatsReserves

Sid	sname	Rating	age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19

Bid	Bname	color
11	Marine	Red
24	Clipper	Blue
33	Wooden	Black
41	Marine	Green

Sid	bid	Day
1	24	2068-08-11
1	11	2068-08-11
1	41	2068-08-22
2	33	2068-11-08
2	11	2068-08-19
11	41	2068-09-23
9	24	2068-08-10
9	11	2069-08-15
9	33	2068-05-21

Now we want to select the content of the columns named "sname" and "age" from the table Sailors. We use the following SELECT statement:

SELECT sname, age
FROM Sailors;

The result-set will look like this:

Sname	Age
Ajaya	33
Robin	43
Ganga	28
Manoj	31
Rahul	22
Sanjaya	42
Raju	19

SELECT * Example

Now we want to select all the columns from the “Sailors" table.We use the following SELECT statement:

*SELECT * FROM Sailors*

p:The asterisk (*) is a quick way of selecting all columns!

The result-set will look like this:

Sid	sname	Rating	age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19

The SQL SELECT DISTINCT Statement:

In a table, some of the columns may contain duplicate values. This is not a problem; however, sometimesyou will want to list only the different (distinct) values in a table.The DISTINCT keyword can be used to return only distinct (different) values.

SQL SELECT DISTINCT Syntax:

SELECT DISTINCT column_name(s)
FROM table_name

SELECT DISTINCT Example:

Now we want to select only the distinct values from the column named "bname" from the table "Boats".We use the following SELECT statement:

SELECT DISTINCT bname FROM Boats

The result-set will look like this:

Bname
Marine
Clipper
Wooden

The WHERE Clause:

The WHERE clause is used to extract only those records that fulfill a specified criterion.

SQL WHERE Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name operator value
```

WHERE Clause Example:

Now we want to select only those Sailors whose age is less than 30 from the table Sailors above.
We use the following SELECT statement:

```
SELECT *
FROM Sailors
WHERE age < 30;
```

The result-set will look like this:

Sid	sname	Rating	age
3	Ganga	32	28
7	Rahul	7	22
11	Raju	4	19

Quotes around Text Fields:

SQL uses single quotes around text values (most database systems will also accept double quotes).Although, numeric values should not be enclosed in quotes.

For text values:

This is correct:

```
SELECT *
FROM Sailors
  WHERE sname='Ajaya';
```

This is wrong:

```
SELECT *
FROM Sailors
   WHERE sname=Ajaya;
```

For Numeric values:

This is correct:

```
SELECT *
FROM Sailors
```

```
        WHERE age=32
This is wrong:SELECT *
        FROM Sailors
        WHERE age='32'
```

Operators Allowed in the WHERE Clause:

With the WHERE clause, the following operators can be used:

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern
IN	If you know the exact value you want to return for at least one of the columns.
AND	And
OR	Or

Note: In some versions of SQL the <> operator may be written as !=

The AND & OR Operators:

The AND & OR operators are used to filter records based on more than one condition.

- The AND operator displays a record if both the first condition and the second condition is true.
- The OR operator displays a record if either the first condition or the second condition is true.

AND Operator Example:

Suppose we want to select only the Sailors with the name equal to "Ajaya" AND the age equal to 33:We use the following SELECT statement:

```
SELECT *
FROM Sailors
WHERE sname='Ajaya' AND age=33;
```

The result-set will look like this:

Sid	sname	rating	age
1	Ajaya	12	33

Example 2: *Find the sids of sailors who have reserved a red boat.*

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid = R.bid AND B.color = `red'
```

The result-set will look like this:

Sid
1

2
9

OR Operator Example:

Now we want to select only the Sailors with the first name equal to "Rahul" OR the rating equal to 9:We use the following SELECT statement:

```
SELECT *
FROM Sailors
WHERE sname='Rahul' OR rating=9;
```

The result-set will look like this:

sid	Sname	Rating	Age
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42

Combining AND & OR:

We can also combine AND and OR (use parenthesis to form complex expressions).

Now we want to select only the Sailors of rating equal to 9 AND the age equal to 31 OR to 42:We use the following SELECT statement:

```
SELECT *
FROM Sailors
WHERE rating=9 AND (age=31 OR age=42)
```

The result-set will look like this:

sid	Sname	Rating	Age
4	Manoj	9	31
9	Sanjaya	9	42

The ORDER BY Keyword:

The ORDER BY keyword is used to sort the result-set by a specified column.The ORDER BY keyword sorts the records in ascending order by default.If you want to sort the records in a descending order, you can use the DESC keyword.

SQL ORDER BY Syntax:

```
SELECT column_name(s)
FROM table_name
ORDER BY column_name(s) ASC|DESC
```

ORDER BY Example:

Now we want to select all the Sailors from the table above, however, we want to sort the Sailors by their name.We use the following SELECT statement:

```
SELECT *
 FROM Sailors
ORDER BY sname;
```

The result-set will look like this:

Sid	Sname	Rating	Age
1	Ajaya	12	33
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
11	Raju	4	19
2	Robin	11	43
9	Sanjaya	9	42

ORDER BY DESC Example:

Now we want to select all the Sailors from the table above, however, we want to sort the Sailors descending by their name.We use the following SELECT statement:

```
SELECT *
FROM Sailors
ORDER BY sname DESC
```

The result-set will look like this:

Sid	Sname	Rating	Age
9	Sanjaya	9	42
2	Robin	11	43
11	Raju	4	19
7	Rahul	7	22
4	Manoj	9	31
3	Ganga	32	28
1	Ajaya	12	33

The SQL BETWEEN Operator:

The BETWEEN operator is used to select values within a range.The values can be numbers, text, or dates.

SQL BETWEEN Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

BETWEEN Operator Example:

The following SQL statement selects all Sailors with age BETWEEN 20 and 40:

```
SELECT *
FROM Sailors
WHERE age BETWEEN 20 AND 40;
```

The result-set will look like this:

Sid	Sname	Rating	Age
7	Rahul	7	22
4	Manoj	9	31
3	Ganga	32	28
1	Ajaya	12	33

NOT BETWEEN Operator Example:

To display the Sailors outside the range of the previous example, use NOT BETWEEN:

```
SELECT *
FROM Sailors
WHERE age NOT BETWEEN 20 AND 40;
```

The result-set will look like this:

Sid	Sname	Rating	Age
9	Sanjaya	9	42
2	Robin	11	43
11	Raju	4	19

SQL IN Operator:

The IN operator allows us to specify multiple values in a WHERE clause.

SQL IN Syntax:

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2...);
```

IN Operator Example:

The following SQL statement selects all Sailors with a rating of 9 or 11:

```
SELECT *
FROM Sailors
WHERE Rating IN (9,11);
```

Its equivalent query by using OR operator is as below:

```
LECT *
ROM Sailors
HERE Rating=9 ORRating=11;
```

The result-set will look like this:

Sid	sname	Rating	Age
2	Robin	11	43
4	Manoj	9	31
9	Sanjaya	9	42

NOT IN Operator Example:

The following SQL statement selects all Sailors with age not 11 or 9:

```
SELECT *
FROM Sailors
WHERE rating NOT IN (11, 9);
```

The result-set will look like this:

Sid	sname	Rating	age
1	Ajaya	12	33
3	Ganga	32	28
7	Rahul	7	22
11	Raju	4	19

String operations:

SQL specifies strings by enclosing them in single quotes, for example 'Pokhara'. The most commonly used operation on strings is pattern matching. It uses the operator ***LIKE.*** We describe the patterns by using two special characters:

- Percent (%): The % character matches any substring, even the empty string.
- Underscore (_): The underscore stands for exactly one character. It matches any character.

To illustrate pattern matching, we consider the following examples:

- 'A_Z': All string that starts with 'A', another character and end with 'Z'. For example, 'ABZ' and 'A2Z' both satisfy this condition but 'ABHZ' does not because between A and Z there are two characters are present instead of one.
- 'ABC%': All strings that start with 'ABC'.
- '%ABC': All strings that ends with 'ABC'.
- '%AN%': All strings that contains the pattern 'AN' anywhere. For example, 'ANGELS' ,'SAN', 'FRANCISCO' etc.
- '_ _ _': matches any strings of exactly three characters.
- '_ _ _%': matches any strings of at least three characters.

Example:

```
SELECT *
FROM Sailors
WHERE sname LIKE '%ya';
```

This SQL statement will match any Sailors first names that end with 'ya'.

The result-set will look like this:

Sid	sname	Rating	age
1	Ajaya	12	33
9	Sanjaya	9	42

Set Operations:

Some time it is useful to combine query results from two or more queries into a single result. SQL supports three set operators which are:

- SQL Union
- SQL Intersection and
- SQL Except (Minus)

These operators have the pattern:

```
<query1><set operator><query2>
```

SQL Union Operation:

In SQL the **UNION** clause combines the results of two SQL queries into a single table of all matching rows. The two queries must result in the same number of columns and compatible data types in order to unite. Any duplicate records are automatically removed unless UNION ALL is used.

Example: ***Find the names of sailors who have reserved a red or a green boat.***

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
AND (B.color = `red' OR B.color = `green')
```

This query is difficult to understand (and also quite inefficient to execute,as it turns out). A better solution for this query is to use UNION as follows:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = `red'
UNION
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = `green'
```

The result-set will look like this:

Sname
Sanjaya
Robin
Raju
Ajaya

UNION ALL gives different results, because it will not eliminate duplicates. Executing this statement:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = `red'
UNION ALL
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = `green'
```

The result-set will look like this:

Sname
Sanjaya
Robin
Raju
Ajaya
Ajaya

INTERSECT Operation:

The SQL INTERSECT operator takes the results of two queries and returns only rows that appear in both result sets.The INTERSECT operator removes duplicate rows from the final result set. The INTERSECT ALL operator does not remove duplicate rows from the final result set.

Example: ***Find the names of sailors who have reserved a red and a green boat.***

```
SELECT S.sname
FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2
WHERE S.sid = R1.sid AND R1.bid = B1.bid AND S.sid = R2.sid AND R2.bid = B2.bid AND
B.color = `red' AND B.color = `green';
```

This query is difficult to understand (and also quite inefficient to execute,as it turns out). A better solution for this query is to use INTERSECT as follows:

The result-set will look like this:

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = `red'
INTERSECT
```

```
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = `green';
```

The result-set will look like this:

Sname
Ajaya

Except Operation:

The SQL EXCEPT operator takes the distinct rows of one query and returns the rows that do not appear in a second result set.

Example: ***Find the names of sailors who have reserved a red boats but not a green boat.***

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = `red'
EXCEPT
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = `green'
```

The EXCEPT operation automatically eliminate duplicates. If we want to retain all duplicates, we must write EXCEPT ALL in place of EXCEPT.

SQL Aggregate Functions:

Aggregate functions are functions that take a collection of values as input and return a single value.

Useful aggregate functions are:

- SUM() - Returns the sum
- AVG() - Returns the average value
- COUNT() - Returns the number of rows
- MAX() - Returns the largest value
- MIN() - Returns the smallest value
- FIRST() - Returns the first value
- LAST() - Returns the last value

Example 1: find sum of rating of all sailors.

```
SELECT SUM (rating)
FROM Sailors;
```

The result-set will look like this:

Sum(rating)
85

Example 2: find average age of all sailors.

```
SELECT AVG (age)
FROM Sailors;
```

The result-set will look like this:

Avg(age)
29.7143

Example 3: find average age of all sailors with a rating of 9.

```
SELECT AVG (age)
FROM Sailors
```

WHERE rating=9;

The result-set will look like this:

Avg(age)
31.5000

Example 4: find name and age of oldest sailor.

```
SELECT sname, age
FROM Sailors
WHERE age = (SELECT MAX(age)
             FROM Sailors);
```

The result-set will look like this:

Sname	max(age)
Robin	43

Example 5: count number of sailors.

```
SELECT COUNT(*)
FROM Sailors;
```

The result-set will look like this:

count(*)
7

Example 6: Find the names of sailors who are older than the oldest sailor with a rating of 9.

```
SELECT S.sname
FROM Sailors S
WHERE S.age>(SELECT MAX (S2.age)
FROM Sailors S2
WHERE S2.rating =9);
```

The result-set will look like this:

Sname
Robin

Example 7: find the maximum and minimum aged sailors name.

```
SELECT sname
FROM Sailors
WHERE age=(SELECT  max(age) FROM Sailors)
UNION
SELECT S1.sname
FROM Sailors S1
WHERE S1.age=(SELECT  min(age) FROM Sailors);
```

The result-set will look like this:

Sname
Robin
Raju

GROUP BY Clause:

The SQL GROUP BY clause is use to divide the rows in a table into groups. The GROUP BY statement is used along with the SQL aggregate functions. In GROUP BY clause, the tuples with same values are placed in one group.

Example: ***Find the age of the youngest sailor for each rating level.***

```
SELECT rating, MIN(age)
FROM Sailors
GROUP BY rating;
```

The result-set will look like this:

Rating	Min(age)
4	19
7	22
9	31
11	43
12	33
32	28

This table display the minimum age of each group according to their rating.

SQL HAVING Clause:

The SQL HAVING clause allows us to specify conditions on the rows for each group. It is used instead of the WHERE clause when Aggregate Functions are used. HAVING clause should follow the GROUP BY clause if we are using it.

Example: let's take an instance S3 of Sailors,

S3

Sid	sname	Rating	age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19
22	Robin	11	54
32	Anish	7	21

Example: Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age>= 18
GROUP BY S.rating
HAVING COUNT (*) >1
```

The result-set will look like this:

Rating	Minage
7	21
9	31
11	43

Example 2: Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.

```
SELECT S.rating, AVG ( S.age )
```

```
FROM Sailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 <(SELECT COUNT (*)
            FROM Sailors S2
            WHERE S.rating = S2.rating );
```

NULL Values:

SQL allows the use of NULL values to indicate absence of information about the value of an attribute. It has a special meaning in the database- the value of the column is not currently known but its value may be known at a later time.

A special comparison operator IS NULL is used to test a column value for NULL. It has following general format:

```
Value1 IS [NOT] NULL;
```

This comparison operator return *true* if value contains NULL, otherwise return *false*. The optional NOT reverses the result.

Following syntax is illegal in SQL:

```
WHERE attribute=NULL;
```

Example: let's take an instance S4 of Sailors,

S4

Sid	sname	Rating	Age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	NULL	NULL
11	Raju	4	19
22	Robin	NULL	NULL
32	Anish	NULL	NULL

Find all sailors that appear in S4 relation with NULL values for rating and age:

```
SELECT sname
FROM S4
WHERE rating IS NULL AND age IS NULL;
```

The result-set will look like this:

sname
Sanjaya
Robin
Anish

Nested Sub-queries:

A nested query is a query that has another query embedded within it; the embedded query is called a sub-query. The result of sub query is used by the main query (outer query). We can place the sub-query in a number of SQL clauses including:

- The WHERE clause
- The HAVING clause

- The FROM clause

A common use of sub-queries is to perform tasks for set membership and make set comparison.

Set Membership:

The IN connective is used to test a set membership, where set is a collection of values produced by SELECT clause in sub-query. The NOT IN connective is used to test for absence of set membership.

Example1: Find the names of sailors who have reserved boat 41.

```
SELECT sname
FROM  Sailors
WHERE sid IN ( SELECT sid
                FROM Reserves
                WHERE bid=41);
```

The result-set will look like this:

Sname
Ajaya
Raju

Example 2: Find the names of sailors who have reserved a red boat.

```
SELECT sname
FROM sailors
WHERE sid IN (SELECT sid
              FROM Reservs
              WHERE bid IN ( SELECT bid
                             FROM Boats
                             WHERE color='Red'));
```

The result-set will look like this:

sname
Ajaya
Robin
Sanjaya

Example 3: Find the names of sailors who have not reserved a red boat.

```
SELECT sname
FROM sailors
WHERE sid NOT IN (SELECT sid
              FROM Reservs
              WHERE bid IN ( SELECT bid
                             FROM Boats
                             WHERE color='Red'));
```

The result-set will look like this:

Sname
Ganga
Manoj
Rahul
Raju

Set Comparison:

The comparison operators are used to compare sets in nested sub-query. SQL allows following set comparisons:

< SOME, <= SOME, > SOME, >= SOME, = SOME, <> SOME

<ALL, <=ALL, >ALL, >=ALL, = ALL, <>ALL

The keyword ANY is synonymous to SOME in SQL.

Example 1: let's take an instance S4 of Sailors as:

S4

Sid	Sname	Rating	Age
1	Ajaya	12	33
2	Robin	11	43
3	Ganga	32	28
4	Manoj	9	31
7	Rahul	7	22
9	Sanjaya	9	42
11	Raju	4	19
8	Rahul	6	76

Find the id and names of sailors whose rating is better than some sailor called "Rahul".

```
SELECT sid, sname
FROM S4
WHERE rating >ANY (SELECT rating
                   FROM S4
                   WHERE sname='Rahul');
```

The result-set will look like this:

Sid	Sname
1	Ajaya
2	Robin
3	Ganga
4	Manoj
7	Rahul
9	Sanjaya

Example 2: Find the id and names of sailors whose rating is better than every sailor called "Rahul".

```
SELECT sid, sname
FROM S4
WHERE rating >ALL (SELECT rating
                   FROM S4
                   WHERE sname='Rahul');
```

The result-set will look like this:

Sid	Sname
1	Ajaya
2	Robin
3	Ganga
4	Manoj
9	Sanjaya

Example 3: Find the id and name of sailorwith height rating.

```
SELECT sid, sname
FROM S4
WHERE rating >=ALL (SELECT rating
                    FROM S4);
```

The result-set will look like this:

Sid	Sname
3	Ganga

Note: IN and NOT IN are equivalent to =ANY and <> respectively.

Views:

A database view is a logical table. It does not physically store data like tables but represent data stored in underlying tables in different formats. A view does not require desk space and we can use view in most places where a table can be used.

Since the views are derived from other tables thus when the data in its source tables are updated, the view reflects the updates as well. They also can be used by DBA to enforce database security.

Advantages of Views:

- Database security: view allows users to access only those sections of database that directly concerns them.
- View provides data independence.
- Easier querying
- Shielding from change
- Views provide group of users to access the data according to their criteria.
- Vies allow the same data to be seen by different users in different ways at the same time.

Syntax for creating view is: ↓ Optional

CREATE VIEW <view name><columns> AS <query expression>

Where, <query expression> is any legal query expression.

Example: Following view contains the id, name, rating+5 and age of those Sailors whose age is greater than 30:

```
CREATE VIEW Sailor_view  AS
SELECT sid, sname, rating+5, age
FROM Sailors
WHERE age>30;
```

Now by executing this query we get following view (logical table);

Sailor_view

Sid	sname	Rating+5	age
1	Ajaya	17	33
2	Robin	16	43
9	Sanjaya	14	42
8	Rahul	11	76

Now any valid database operations can be performed in this view like in that of general table.

Modification of the database:

Until now we only study about how information can be extract from the database. Now, we show how to add, remove, or change information with SQL.

To modify database, there are mainly three operations are used:

- ***Insertion***
- ***Deletion and***
- ***Updates***

1.Insertion:

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted.

Example 1: suppose we need to insert a new record of Sailors of id is11, name is "Rahul", rating is 9 and of age is 29 then we write following SQL query,

```
INSERT INTO Sailors
        VALUES(11,'Rahul',9,29);
                OR
INSERT INTO Sailors (sid, sname, rating, age)
        VALUES (11,'Rahul',9,29);
```

More generally, we might want to insert tuples on the basis of the result of query.

Example 2:suppose we have already some tuples on the relation 'Sailores'. Suppose we need to insert those tuples of sailors into their own relation whose rating is less than 7, this can be write as,

```
INSERT INTO Sailors
        SELECT *
        FROM Sailors
        WHERE rating<7;
```

2. Deletion:

It is used to remove whole records or rows from the table.

Syntax:

```
DELETE FROM table_name
 WHERE <predicate>
```

Example 1: suppose we need to remove all tuples of Sailors whose age is 32,

```
DELETE FROM Sailors
WHERE age=32;
```

Example 2: Remove all tuples of Sailors whose age is less than 30 and rating greater than 7,

```
DELETE FROM Sailors
WHERE age<32 AND rating>7;
```

3. Updates:

If we need to change a particular value in a tuple without changing all values in the tuple, then for this purpose we use update operation.

Syntax:

UPDATE table_name
SET <column i> = <expression i>;

Example: suppose we need to increase the rating of those sailors whose age is greater than 40 by 20%, this can be write as,

UPDATE sailors
*SET rating=rating+rating*0.2*
WHERE age>40;

Joined relations:

An SQL JOIN clause is used to combine rows from two or more tables, based on a common field between them.

The types the different SQL JOINs are:

- **INNER JOIN**
- **LEFT OUTER JOIN**
- **RIGHT OUTER JOIN**
- **FULL OUTER JOIN**

INNER JOIN:

It is most common type of join. An SQL INNER JOIN return all rows from multiple tables where the join condition is met.

SQL INNER JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
INNER JOIN *table2*
ON *table1.column_name=table2.column_name*;

Note: INNER JOIN is the same as JOIN.

Example 1: Find the sailor id, boat id, boat name, boat color of those sailors who have reserved a red boat.

SELECT sailors.sid, boats.bid, boats.bid, boats.bname, boats.color
FROM sailors INNER JOIN reserver INNER JOIN boats
WHERE sailors.sid=reserver.sid AND reserver.bid=boats.bid;

The result-set will look like this:

sid	sname	bid	bname	color
1	Ajaya	24	Clipper	Blue
1	Ajaya	11	Marine	Red
1	Ajaya	41	Marine	Green
2	Robin	33	Wooden	Black
2	Robin	11	Marine	Red
11	Raju	41	Marine	Green
9	Sanjaya	24	Clipper	Blue
9	Sanjaya	11	Marine	Red
9	Sanjaya	33	Wooden	Black

Example 2: Find the name and age of those sailors who have reserved a Marine boat.

SELECT sailors.sname, sailors.age
FROM sailors INNER JOIN reserver INNER JOIN boats
WHERE sailors.sid=reserver.sid AND reserver.bid=boats.bid;

The result-set will look like this:

sname	age
Ajaya	33
Ajaya	33
Ajaya	33
Robin	43
Robin	43
Raju	19
Sanjaya	42
Sanjaya	42
Sanjaya	42

LEFT OUTER JOIN:

The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2). The result is NULL in the right side when there is no match.

SQL LEFT JOIN Syntax:

SELECT *column_name(s)*
FROM *table1*
LEFT OUTER JOIN *table2*
ON *table1.column_name=table2.column_name*;

RIGHT OUTER JOIN:

The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1). The result is NULL in the left side when there is no match.

SQL RIGHT JOIN Syntax:

SELECT column_name(s)
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name=table2.column_name;

FULL OUTER JOIN:

The FULL OUTER JOIN keyword returns all rows from the left table (table1) and from the right table (table2).The FULL OUTER JOIN keyword combines the result of both LEFT and RIGHT joins.

SQL FULL OUTER JOIN Syntax:

SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;

Data definition languages:

The DDL part of SQL permits database tables to be created or deleted. It also defines indexes (keys), specify links between tables, and impose constraints between tables.

The most important DDL statements in SQLare:

- **CREATE DATABASE**- creates a new database
- **ALTER DATABASE**- modifies a database
- **CREATE TABLE**- creates a new table
- **ALTER TABLE**- modifies a table

- **DROP TABLE**- deletes a table
- **CREATE INDEX**- creates an index (search key)
- **DROP INDEX**- deletes an index

Domain type (data type) in SQL:

When we create a table each column of the table must be specified by their domain or data type. Due to e it helps us what type of data will be stored in the field.

The SQL standard supports a variety of build-in domain types which are:

- **Char (n):** A fixed length character data (string). Also we can use full form ***character***.
- **Varchar(n):** A variable character string.
- **Int:** used to represent whole number. Also we can use it's full form ***integer.***
- **Numeric(p, d):** A fixed point number with user-specified precision. The number consists of p digits (plus a sign), and d represent the number of digits to right of decimal point.
- **Real, double precision:** floating point numbers with machine dependent precisions.
- **Float(n):** floating point number with precision of at least n digits.
- **Date:** A calendar date containing a four digit year, month and day. Eg '2006-04-22'
- **Time:** The time of day, in hours, minutes, and seconds.eg '09:34:23'
- **Timestamp:** combination of date and time.eg '2008-05-21 11:23:08'

CREATE DATABASE:

The ***CREATE DATABASE***statement is used to create a database.

Syntax:

CREATE DATABASE *dbname*;

Example:

CREATE DATABASE *my_db*;

After creating database 'my_db' we need to connect it as;

CONNECT my_db;

ALTER DATABASE:

Allow us to modify existing database name.

Syntax:

CREATE TABLE:

Allow us to create a new table within given database.

Syntax:

```
CREATE TABLE <table_name>
(
        <column1>    <data type> [not null] [unique][<integrity constraint>],
        <column2>    <data type> [not null] [unique][<integrity constraint>],
        …….
        ……
        <column n>   <data type> [not null] [unique][<integrity constraint>]
)
```

Note: [ ]: optional

Example:

```
CREATE TABLE Sailors
(
        Sid     INTEGER NOT NULL,
        Sname VARCHAR(12),
        Rating INTEGER,
        Age     INTEGER,
        PRIMARY KEY (sid)
)
```

ALTER TABLE:

Allow us to modifya given table.

The structure of given table can be changed either of the following:

- By adding new column in existing table
- By deleting some columns from an existing table and
- By modifying some columns of given table

A new column can be added to the table as follows:

Syntax:

ALTER TABLE <table name>
ADD (<column_name><datatype>);

Example: suppose we want to add a new column 'addresses' to an existing table Sailors,

ALTER TABLE Sailors
ADD (addresses varchar(15));

An existing column can be removed from the table as,

ALTER TABLE <table_name>
DROP (column_name);

Example: suppose we want to remove an existing column 'addresses' from the table sailors as,

ALTER TABLE Sailors
DROP (addresses);

An existing column can be modified as,

ALTER TABLE <table_name>
MODIFY (<column name><data type>);

Example: modify sailors relation by changing the range of the name of sailors by 20,

ALTER TABLE sailors
MODIFY (snamevarchar(20));

DROP TABLE

It allows us to remove an existing table from the database.

Syntax:

DROP TABLE <table name>

Example: if we want to remove a table ‘Sailors’ from the database my_db as,

DROP TABLE Sailors;

Embedded SQL:

The programming language in which SQL queries are embedded is called host language. And the SQL structures permitted in the host language is called embedded SQL. They are compiled by the embedded SQL processor.

Writing queries in SQL is usually much easier than coding same query in a programming language. However, a programmer must access to a database from a general purpose programming language for following two reasons:

- Not all queries can be expressed in SQL
- Non-declarative actions such as printing a report, interacting with user, or sending the result of query to a graphical user interface etc. cannot be done from the SQL.

Dynamic SQL:

The *dynamic SQL* component of SQL allows programs to construct and submit SQLqueries at run time. In contrast, embedded SQL statements must be completely presentat compile time; they are compiled by the embedded SQL preprocessor.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son

SQL Practice:-

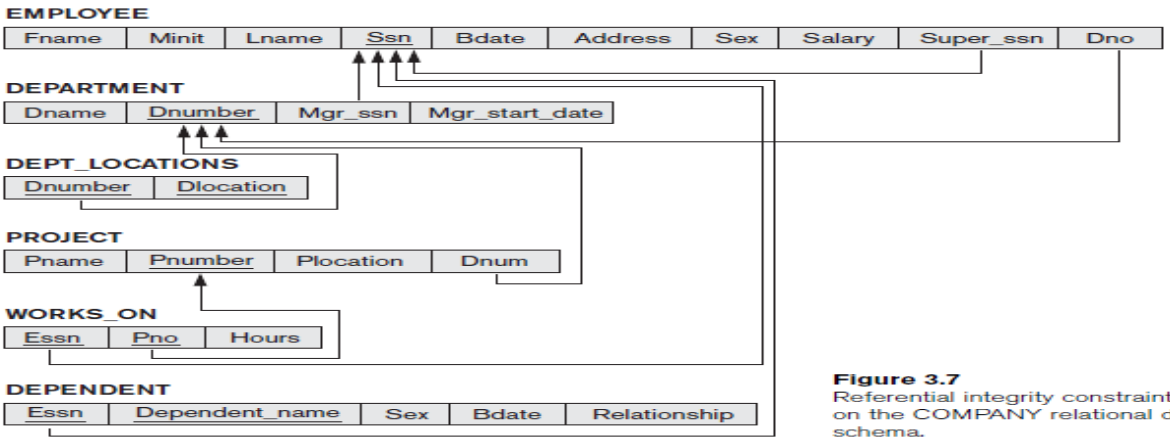


Figure 3.7
Referential integrity constraints displayed on the COMPANY relational database schema.

```
CREATE TABLE EMPLOYEE
        ( Fname             VARCHAR(15)             NOT NULL,
          Minit             CHAR,
          Lname             VARCHAR(15)             NOT NULL,
          Ssn               CHAR(9)                 NOT NULL,
          Bdate             DATE,
          Address           VARCHAR(30),
          Sex               CHAR,
          Salary            DECIMAL(10,2),
          Super_ssn         CHAR(9),
          Dno               INT                     NOT NULL,
        PRIMARY KEY (Ssn),
        FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Dno) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE DEPARTMENT
        ( Dname             VARCHAR(15)             NOT NULL,
          Dnumber           INT                     NOT NULL,
          Mgr_ssn           CHAR(9)                 NOT NULL,
          Mgr_start_date    DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber           INT                     NOT NULL,
          Dlocation         VARCHAR(15)             NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname             VARCHAR(15)             NOT NULL,
          Pnumber           INT                     NOT NULL,
          Plocation         VARCHAR(15),
          Dnum              INT                     NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn              CHAR(9)                 NOT NULL,
          Pno               INT                     NOT NULL,
          Hours             DECIMAL(3,1)            NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn              CHAR(9)                 NOT NULL,
          Dependent_name    VARCHAR(15)             NOT NULL,
          Sex               CHAR,
          Bdate             DATE,
          Relationship      VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

Figure 4.1
SQL CREATE TABLE data definition statements for defining the COMPANY schema from Figure 3.7.

Query 1. Retrieve the name and address of all employees who work for the ‘Research’ department.

Q1: SELECT Fname, Lname, Address FROM EMPLOYEE, DEPARTMENT
WHERE Dname=‘Research’ AND Dnumber=Dno;

Query 2. For every project located in ‘Stafford’, list the project number, the controlling department number, and the department manager’s last name, address, and birth date.

```
Q2:     SELECT      Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE       Dnum=Dnumber AND Mgr_ssn=Ssn AND
Plocation='Stafford';
```

Query 8. For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

```
Q8:     SELECT      E.Fname, E.Lname, S.Fname, S.Lname

FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE       E.Super_ssn=S.Ssn;
```

Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:     SELECT      Ssn
        FROM EMPLOYEE;
Q10:    SELECT      Ssn, Dname
        FROM EMPLOYEE, DEPARTMENT;
```

Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:    SELECT      ALL Salary
        FROM EMPLOYEE;
Q11A:   SELECT      DISTINCT Salary
        FROM EMPLOYEE;
```

Query 4. Make a list of all project numbers for projects that involve an employee whose last name is 'Smith', either as a worker or as a manager of the department that controls the project.

```
( SELECT DISTINCT Pnumber
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE  Dnum=Dnumber AND Mgr_ssn=Ssn
AND Lname='Smith' )
UNION
( SELECT DISTINCT Pnumber

FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE Pnumber=Pno AND Essn=Ssn
 AND Lname='Smith' );
```

Query 12. Retrieve all employees whose address is in Houston, Texas.

```
SELECT      Fname, Lname
```

FROM EMPLOYEE

WHERE Address LIKE '%Houston,TX%';

Query 12A. Find all employees who were born during the 1950s.

SELECT Fname, Lname

FROM EMPLOYEE

WHERE Bdate LIKE '_ _ 5 _ _ _ _ _ _ _';

Query 13. Show the resulting salaries if every employee working on the 'ProductX' project is given a 10 percent raise.

SELECT E.Fname, E.Lname, 1.1 * E.Salary AS Increased_sal
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE E.Ssn=W.Essn AND W.Pno=P.Pnumber AND P.Pname='ProductX';

Query 14. Retrieve all employees in department 5 whose salary is between $30,000 and $40,000.

Q14: SELECT *

FROM EMPLOYEE

WHERE (Salary BETWEEN 30000 AND 40000) AND Dno = 5;

Query 15. Retrieve a list of employees and the projects they are working on, ordered by department and, within each department, ordered alphabetically by last name, then first name.

Q15: SELECT D.Dname, E.Lname, E.Fname, P.Pname
FROM DEPARTMENT D, EMPLOYEE E, WORKS_ON W,

PROJECT P

WHERE D.Dnumber= E.Dno AND E.Ssn= W.Essn AND
W.Pno= P.Pnumber

ORDER BY D.Dname, E.Lname, E.Fname;

Insert update delete command:-

INSERT INTO EMPLOYEE VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30', '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);

```
DELETE FROM       EMPLOYEE
WHERE       Lname='Brown';
DELETE FROM       EMPLOYEE
WHERE       Ssn='123456789';
DELETE FROM       EMPLOYEE
WHERE       Dno=5;
DELETE FROM       EMPLOYEE;
```

give all employees in the 'Research' department a 10 percent raise in salary

```
UPDATE  EMPLOYEE
SET Salary = Salary * 1.1
WHERE  Dno = 5;
```

Query 18. Retrieve the names of all employees who do not have supervisors.

```
Q18:    SELECT       Fname, Lname

FROM EMPLOYEE

WHERE       Super_ssn IS NULL;
```

Query 16. Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
Q16:    SELECT       E.Fname, E.Lname

FROM EMPLOYEE AS E

WHERE       E.Ssn IN       (  SELECT     Essn

FROM DEPENDENT AS D

WHERE       E.Fname=D.Dependent_name

AND E.Sex=D.Sex );
```

Query 6. Retrieve the names of employees who have no dependents.

```
Q6:     SELECT       Fname, Lname

FROM EMPLOYEE

WHERE       NOT EXISTS ( SELECT      *
```

```
FROM DEPENDENT

WHERE       Ssn=Essn );
```

Query 7. List the names of managers who have at least one dependent.

```
Q7:   SELECT      Fname, Lname
      FROM EMPLOYEE
      WHERE       EXISTS ( SELECT    *
            FROM DEPENDENT
            WHERE       Ssn=Essn )
            AND
            EXISTS ( SELECT    *
            FROM DEPARTMENT
            WHERE       Ssn=Mgr_ssn );
```

Query 17. Retrieve the Social Security numbers of all employees who work on project numbers 1, 2, or 3.

```
Q17:  SELECT      DISTINCT Essn

FROM WORKS_ON

WHERE       Pno IN (1, 2, 3);
```

Consider query Q1, which retrieves the name and address of every employee who works for the 'Research' department.

```
SELECT Fname, Lname, Address

FROM  (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)

WHERE  Dname='Research';
```

Query 19. Find the sum of the salaries of all employees, the maximum salary, the minimum salary, and the average salary.

```
Q19:  SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM EMPLOYEE;
```

Query 20. Find the sum of the salaries of all employees of the 'Research' department, as well as the maximum salary, the minimum salary, and the aver-age salary in this department.

```
Q20:  SELECT      SUM (Salary), MAX (Salary), MIN (Salary), AVG (Salary)
FROM (EMPLOYEE JOIN DEPARTMENT ON Dno=Dnumber)
```

WHERE Dname='Research';

Queries 21 and 22. Retrieve the total number of employees in the company (Q21) and the number of employees in the 'Research' department (Q22).

```
Q21: SELECT   COUNT (*)
     FROM EMPLOYEE;
Q22: SELECT   COUNT (*)
     FROM EMPLOYEE, DEPARTMENT
     WHERE    DNO=DNUMBER AND DNAME='Research';
```

Query 23. Count the number of distinct salary values in the database.

```
Q23: SELECT   COUNT (DISTINCT Salary)
FROM EMPLOYEE;
```

Query 24. For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
Q24: SELECT   Dno, COUNT (*), AVG (Salary)

FROM EMPLOYEE

GROUP BY  Dno;
```

Query 25. For each project, retrieve the project number, the project name, and the number of employees who work on that project.

```
Q25: SELECT   Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON
WHERE    Pnumber=Pno

GROUP BY  Pnumber, Pname;
```

Query 26. For each project on which more than two employees work, retrieve the project number, the project name, and the number of employees who work on the project.

```
SELECT Pnumber, Pname, COUNT (*)

FROM PROJECT, WORKS_ON
WHERE    Pnumber=Pno

GROUP BY Pnumber, Pname

HAVING   COUNT (*) > 2;
```

Query 27. For each project, retrieve the project number, the project name, and the number of employees from department 5 who work on the project.

```
Q27:  SELECT      Pnumber, Pname, COUNT (*)
FROM PROJECT, WORKS_ON, EMPLOYEE

WHERE      Pnumber=Pno AND Ssn=Essn AND Dno=5

GROUP BY   Pnumber, Pname;
```

Query 28. For each department that has more than five employees, retrieve the department number and the number of its employees who are making more than $40,000.

```
Q28:  SELECT      Dnumber, COUNT (*)
FROM DEPARTMENT, EMPLOYEE

WHERE      Dnumber=Dno AND Salary>40000 AND
( SELECT   Dno

FROM EMPLOYEE

GROUP BY Dno

HAVING     COUNT (*) > 5)
```

Specification of Views in SQL

```
CREATE VIEW      WORKS_ON1
AS SELECT  Fname, Lname, Pname, Hours
FROM EMPLOYEE, PROJECT, WORKS_ON
WHERE      Ssn=Essn AND Pno=Pnumber;
CREATE VIEW      DEPT_INFO(Dept_name, No_of_emps, Total_sal)
AS SELECT  Dname, COUNT (*), SUM (Salary)
FROM DEPARTMENT, EMPLOYEE
WHERE      Dnumber=Dno
GROUP BY   Dname;
```

To retrieve the last name and first name of all employees who work on the ‘ProductX’ project, we can utilize the WORKS_ON1 view

```
SELECT     Fname, Lname

FROM WORKS_ON1

WHERE      Pname=‘ProductX’;
```

To Drop View:-
DROP VIEW WORKS_ON1;

Exercise 1: Consider the following relations:

Student(*snum:* integer, *sname:* string, *major:* string, *level:* string, *age:* integer)
Class(c*name:* string, *meets-at:* time, *room:* string, *fid:* integer)
Enrolled(*snum:* integer, *cname:* string)
Faculty(fid: integer, *fname:* string, *deptid:* integer)

1. Find the names of all Juniors (Level = JR) who are enrolled in a class taught by I. Teach.
2. Find the age of the oldest student who is either a History major or is enrolled in a course taught by I. Teach.
3. Find the names of all classes that either meet in room R128 or have five or more students enrolled.
4. Find the names of all students who are enrolled in two classes that meet at the same time.
5. Find the names of faculty members who teach in every room in which some class is taught.
6. Find the names of faculty members for whom the combined enrollment of the courses that they teach is less than _ve.
7. Print the Level and the average age of students for that Level, for each Level.
8. Print the Level and the average age of students for that Level, for all Levels except JR.
9. Find the names of students who are enrolled in the maximum number of classes.
10. Find the names of students who are not enrolled in any class.

Exercise 2 Consider the following schema:
Suppliers(*sid:* integer, *sname:* string, *address:* string)
Parts(*pid:* integer, *pname:* string, *color:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)
Write the followingqueries in SQL:

1. Find the *pname*s of parts for which there is some supplier.
2. Find the *sname*s of suppliers who supply every part.
3. Find the *sname*s of suppliers who supply every red part.
4. Find the *pname*s of parts supplied by Acme Widget Suppliers and by no one else.
5. Find the *sid*s of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).
7. Find the *sid*s of suppliers who supply only red parts.
8. Find the *sid*s of suppliers who supply a red part and a green part.
9. Find the *sid*s of suppliers who supply a red part or a green part.

Exercise 3Consider the following schema of the relational database
Books(Bid,Btitle, Bauthor, Bpublisher, Bprice)
Members(Member_id, Name, Designation, Age)

Reserve(Member_id,Bid, Date)

1. Create the tables using Books, Members and Reserve by specifying the Primary key, Not NULL , Foreign key Constraints DDL Statement in MySQL database
2. Write SQL DML statement to insert any five tuples (Five records) in each relation(table)
3. Find the Books of Database System title and price above 500.
4. List the books published by TaTa McGraw Hill publication
5. Find the Name of the member who made reserve book in 12-10-2011

Exercise 4Consider the following schema of the relational database

Department(dept_no,d_name, city)
Employee(emp_Id, e_name, salary)
Works(dept_no, emp_Id)

1. Create the above tables by specifying the Primary key, Not NULL , Foreign key Constraints DDL Statement in MySQL database
2. Write DML Statement to Insert any five records in each tables
3. Display the name of the employees.
4. Find the name of the employees whose salary is greater than 10000.
5. Find the department (d_name) of the employee '**Binek'**.

Exercise 5: Consider the following insurance database, where primary keys are underlined:

Teacher (Tid, Tname, Address, Age)
Student (Sid,Sname, Age, sex)
Takes (sid,course-id)
Course (course-id,course_name, text_book)
Teaches(Tid, Cousrse-id)
Taught-by{Sid, Tid}

Construct the following RA expressions for this relational database

a. Fine name, age and sex of all students who takes course "DBMS"
b. Find total number of students who are taught by the teacher "T01"
c. List all course names text books taught by teaher "T16"
d. Find average age of teachers for each course.
e. Insert the record of new teacher "T06" named "Bhupi" of age 27 into database who lives in "Balkhu" and takes course "DBMS"

Employee Database

---

employee (employee_name, street, city)

works (employee_name, company_name, salary)

company (company_name, city)

manages (employee_name, manager_name)

Consider the employee database , where the primary keys are underlined. Give an expression in SQL for each of the following queries.

a. Find the names of all employees who work for First Bank Corporation.

b. Find all employees in the database who live in the same cities as the companies for which

they work.

c. Find all employees in the database who live in the same cities and on the

same streets as do their managers.

d. Find all employees who earn more than the average salary of all employees of their company.

e. Find the company that has the smallest payroll.

Answer: 3

a. Find the names of all employees who work for First Bank Corporation.

```
select employee_name
 from works
 where company_name = 'First Bank Corporation';
```

b. Find all employees in the database who live in the same cities as the companies for which they work.

```
select e.employee_name
from employee e, works w, company c
where e.employee_name = w.employee_name and e.city = c.city and
     w.company_name = c.company_name
```

c. Find all employees in the database who live in the same cities and on the same streets as do their managers.

```
select P.employee_name
from employee P, employee R, manages M
where P.employee_name = M.employee_name and
```

M.manager_name = R.employee_name and

P.street = R.street and P.city = R.city

d. Find all employees who earn more than the average salary of all employees of their company.

The following solution assumes that all people work for at most one company.

```
select employee_name
from works
where salary > (select avg (salary)
                from works )
)
```

e. Find the company that has the smallest payroll.

```
select company name
from works
group by company name
having sum (salary) <= all (select sum (salary)
                            from works
                            group by company name)
```

Consider the relational database . Give an expression in SQL for each of the following queries.

a. Give all employees of First Bank Corporation a 10 percent raise.

b. Give all managers of First Bank Corporation a 10 percent raise.

c. Delete all tuples in the works relation for employees of Small Bank Corpora-

Answer:

a. Give all employees of First Bank Corporation a 10-percent raise. (the solution assumes that each person works for at most one company.)

```
update works
set salary = salary * 1.1
where company name = 'First Bank Corporation'
```

b. Give all managers of First Bank Corporation a 10-percent raise.

```
update works
set salary = salary * 1.1
where employee_name in (select manager_name
                        from manages)
    and company name = 'First Bank Corporation'
```

c. Delete all tuples in the works relation for employees of Small Bank Corporation.

```
delete from works
where company_name = 'Small Bank Corporation'.
```

Consider the relational database . Using SQL, define a view consisting of manager name and the average salary of all employees who work for that manager. Explain why the database system should not allow updates to be expressed in terms of this view.

Answer:

```
create view salinfo as
    select manager name, avg(salary)
    from manages m, works w
    where m.employee name = w.employee name
    group by manager name
```

Updates should not be allowed in this view because there is no way to determine how to change the underlying data. For example, suppose the request

is “change the average salary of employees working for Smith to $200”. Should everybody who works for Smith have their salary changed to $200? Or should the first (or more, if necessary) employee found who works for Smith have their salary adjusted so that the average is $200? Neither approach really makes sense.

Give an SQL schema definition for the employee database of Figure Choose an appropriate domain for each attribute and an appropriate primary key for each relation schema.

Answer:

```
create domain      company names char(20)
create domain      city names char(30)
create domain      person names char(20)

create table       employee
(employee name     person names,
 street            char(30),
 city              city names,
 primary key       (employee name))

create table       works
(employee name     person names,
 company name      company names,
 salary            numeric(8, 2),
 primary key       (employee name))
```

create table company

(company name company names,

city city names,

primary key (company name))

create table manages

(employee name person names,

manager name person names,

primary key (employee name))

Consider the following relational schema

employee(empno, name, office, age)

books(isbn, title, authors, publisher)

loan(empno, isbn, date)

Write the following queries in SQL.

a. Print the names of employees who have borrowed any book published by McGraw-Hill.

b. Print the names of employees who have borrowed all books published by McGraw-Hill.

c. For each publisher, print the names of employees who have borrowed more than five books of that publisher.

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)

Catalog(sid: integer, pid: integer, cost: real)

Relational Algebra and Calculus 29

1. Find the names of suppliers who supply some red part.

```
SELECT    S.sname
      FROM      Suppliers S, Parts P, Catalog C
      WHERE     P.color='red' AND    C.pid=P.pid AND     C.sid=S.sid
```

2. Find the sids of suppliers who supply some red or green part.

```
SELECT    C.sid
      FROM      Catalog C, Parts P
      WHERE     (P.color = 'red' OR  P.color = 'green')
            AND  P.pid = C.pid
```

3. Find the sids of suppliers who supply some red part or are at 221 Packer Street.

```
SELECT    S.sid
      FROM      Suppliers S
      WHERE     S.address = '221 Packer street'
            OR S.sid IN  ( SELECT   C.sid
                  FROM     Parts P, Catalog C
                  WHERE    P.color='red' AND   P.pid = C.pid )
```

4. Find the sids of suppliers who supply some red part and some green part.

```
SELECT    C.sid
      FROM     Parts P, Catalog C
```

```
    WHERE    P.color = 'red' AND   P.pid = C.pid
        AND   EXISTS   ( SELECT  P2.pid
                FROM     Parts P2, Catalog C2
                WHERE    P2.color = 'green' AND    C2.sid = C.sid
                    AND  P2.pid = C2.pid )
```

5. Find the sids of suppliers who supply every part.

```
SELECT   C.sid
    FROM     Catalog C
    WHERE    NOT  EXISTS   (SELECT  P.pid
                FROM     Parts P
                WHERE    NOT  EXISTS   (SELECT   C1.sid
                            FROM     Catalog C1
                            WHERE    C1.sid = C.sid
                                AND  C1.pid = P.pid))
```

6. Find the sids of suppliers who supply every red part.

```
SELECT   C.sid
    FROM     Catalog C
    WHERE    NOT  EXISTS   (SELECT  P.pid
                FROM     Parts P
                WHERE    P.color = 'red'
                AND  (NOT  EXISTS   (SELECT   C1.sid
                            FROM     Catalog C1
                            WHERE    C1.sid = C.sid AND
                                C1.pid = P.pid)))
```

7. Find the sids of suppliers who supply every red or green part.

```
SELECT    C.sid
        FROM      Catalog C
        WHERE    NOT   EXISTS    (SELECT  P.pid
                        FROM     Parts P
                        WHERE    (P.color = 'red' OR    P.color = 'green')
                        AND  (NOT   EXISTS   (SELECT    C1.sid
                                        FROM      Catalog C1
                                        WHERE     C1.sid = C.sid AND
                                              C1.pid = P.pid)))
```

8. Find the sids of suppliers who supply every red part or supply every green part.

```
SELECT   C.sid
      FROM     Catalog C
      WHERE    (NOT  EXISTS   (SELECT  P.pid
                      FROM     Parts P
                      WHERE    P.color = 'red' AND
                      (NOT  EXISTS   (SELECT   C1.sid
                                FROM     Catalog C1
                                WHERE    C1.sid = C.sid AND
                                      C1.pid = P.pid))))
           OR  ( NOT  EXISTS   (SELECT  P1.pid
                        FROM     Parts P1
                        WHERE    P1.color = 'green' AND
                        (NOT  EXISTS   (SELECT   C2.sid
                                  FROM     Catalog C2
                                  WHERE     C2.sid = C.sid AND
```

C2.pid = P1.pid))))

9. Find pairs of sids such that the supplier with the first sid charges more for some part than the supplier with the second sid.

```
SELECT   C1.sid, C2.sid
        FROM     Catalog C1, Catalog C2
        WHERE    C1.pid = C2.pid AND     C1.sid =
C2.sid
              AND  C1.cost > C2.cost
```

10. Find the pids of parts supplied by at least two different suppliers.

```
SELECT   C.pid
        FROM     Catalog C
        WHERE    EXISTS   (SELECT   C1.sid
                  FROM     Catalog C1
                  WHERE    C1.pid = C.pid AND     C1.sid =
C.sid )
```

11. Find the pids of the most expensive parts supplied by suppliers named Yosemite Sham.

```
SELECT    C.pid
        FROM      Catalog C, Suppliers S
        WHERE     S.sname = 'Yosemite Sham' AND         C.sid = S.sid
              AND  C.cost ≥ ALL     (Select C2.cost
                        FROM    Catalog C2, Suppliers S2
                        WHERE S2.sname = 'Yosemite Sham'
                             AND  C2.sid = S2.sid)
```

12. Find the pids of parts supplied by every supplier at less than $200. (If any supplier

either does not supply the part or charges more than $200 for it, the part is not

selected.)

Exercise 6: *employee(employee-name, street, city)*
works(employee-name, company-name, salary)
company(company-name,city)
manages(employee-name, manager-name)

Give an expression in SQL for each of the following queries:

a) Find the names, street address, and cities of residence for all employees who work for 'First Bank Corporation' and earn more than $10,000.

```
select employee.employee-name, employee.street, employee.city from employee,
    works
where employee.employee-name=works.employee-name
        and company-name = 'First Bank Corporation' and salary > 10000)
```

b) Find the names of all employees in the database who live in the same cities as the companies for which they work.

```
select e.employee-name
from employee e, works w, company c
    where e.employee-name = w.employee-name and e.city = c.city and
          w.company-name = c.company-name
```

c) Find the names of all employees in the database who live in the same cities and on the same streets as do their managers.

```
select p.employee-name
from employee p, employee r, manages m
where p.employee-name = m.employee-name and m.manager-name = r.employee-
    name
        and p.street = r.street and p.city = r.city
```

d) Find the names of all employees in the database who do not work for 'First Bank Corporation'. Assume that all people work for exactly one company.

```
select employee-name from works
where company-name <> 'First Bank Corporation'
```

e) Find the names of all employees in the database who earn more than every employee of 'Small Bank Corporation'. Assume that all people work for at most one company.

```
select  employee-name  from
   works
   where salary > all (select salary from
                        works
                  where company-name = 'Small Bank Corporation')
```

f) Assume that the companies may be located in several cities. Find all companies located in every city in which 'Small Bank Corporation' is located.

```
select  s.company-name  from
   company s where not exists
((select city from company where company-name = 'Small Bank
   Corporation')
except
(select city from company t where s.company-name = t.company-name))
```

g) Find the names of all employees who earn more than the average salary of all employees of their company. Assume that all people work for at most one company.

```
select  employee-name  from
   works t
where salary >(select avg(salary) from works s
               where t.company-name = s.company-name)
```

h) Find the name of the company that has the smallest payroll.

```
select  company-name  from
   works
group by company-name
having sum(salary) <= all (select sum(salary)
                           from works
                                  group by company-name)
```

Integrity Constraints

Unit 5

- Domain constraints
- Referential constraints
- Triggers
- Assertion

Integrity constraints ensure that changes made to the database by authorized usersdo not result in a loss of data consistency. Thus, integrity constraints guard againstaccidental damage to the database.

E-R model ensure two types of integrity constraints:

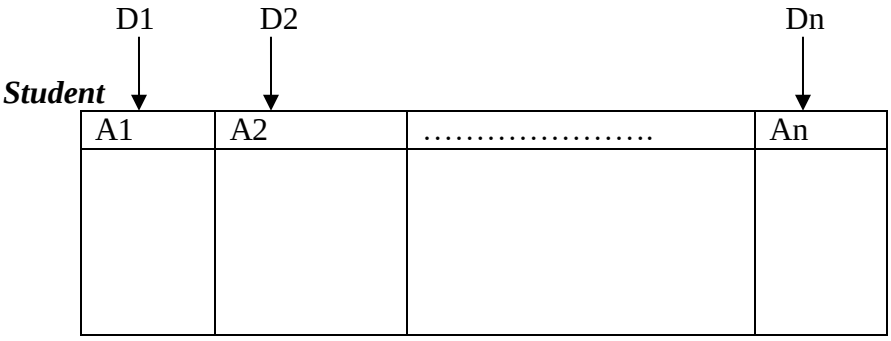
– **Key declaration**: Primary and candidate

– **Form of relationship**: many to many, one to many, one to one

Types of constraints:

Domain constraint:

Domain is a pool of values of the same type from which one or more attributes in one or more tables take their values.



In the above student table, the attribute A1 draws value from domain D1, A2 from D2 and so on.

Domain integrity means the definition of a valid set of values for an attribute. You define

- ✓ data type
- ✓ Length or size
- ✓ Is null value allowed
- ✓ Is the value unique or notfor an attribute.

Domain constraints are the most elementary form of integrity constraint. They are tested essentially by the system whenever a new data item is entered into the database.

It is possible for several attributes to have the same domain.

- ✓ For example, the attributes *customer-name* and *employee-name* might have the same domain: ***the set of all person names***.
- ✓ However, the domains of *balance* and *branch-name* certainly ought to be distinct.
- ✓ It is perhaps less clear whether *customer-name* and *branch-name* should have the same domain.

At the implementation level, both customer names and branch names are character strings. However, we would normally not consider the query "Find all customers who have the same name as a branch" to be a meaningful query. Thus, if we view the database at the conceptual, rather than the physical, level, *customer-name* and *branch-name* should have distinct domains.

The **CREATE DOMAIN**clause can be used to define new domains. For example, to ensure that *rating* mustbe an integer in the range 1 to 10, we could use:

CREATE DOMAIN RATINGVAL INTEGER DEFAULT 0
CHECK (VALUE>= 1 AND VALUE <= 10)

Entity Integrity

The entity integrity constraint ensures that the primary key of a relation must be unique and not null.

Example: ***Employee***

cid	Cname	caddress	Cphone
1	Abin	Kathmandu	9849248488
2	Anish	Lalitpur	9849245544
?	Binek	Kirtipur	9813334849

Entity integrity violation

Referential Integrity

Referential integrity ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation to establish the relationship between tables.

For referential integrity to hold in a relational database, any field in a table that is declared a foreign key can contain either a null value, or only values from a parent table's primary key. For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity.

In relational model we often store data in different tables and put them together to get complete example. For example, in PAYMENTS table we have only ROLLNO of the student. To get remaining information about the student we have to use STUDENTS table.

STUDENTS

RollNO	Name	Address
1	Anisha	Ktm
2	Bibek	Pokhara
3	Nikey	Lalitpur
4	Rashmi	Bhaktapur

PAYMENTS

RollNO	Date	Amount
1	12-03-2010	10000
3	06-08-2011	5000
2	02-07-2012	9000
4	14-03-2013	15000

reign Key

Thus for referential integrity a foreign key can have only two possible values- either the relevant primary key or a null value. No other values are allowed.

Referential Integrity in SQL

Primary and candidate keys and foreign keys can be specified as parts of the SQL **create table** statement:

Example:

CREATE TABLE*Sailors*
(
*sid**integer not null**,*
*sname**varcha**r*(20)**,**
rating **integer**,
*age **integer**,*
primary key (sid*)*
)

CREATE TABLE*Boats*
(
bid**integer not null,**
bname**varchar**(20),
color**varchar**(10),
primary key *(bid)*
)

CREATE TABLE*Reserve*
(
sid**integer,**
bid **integer,**
rdate**date**,
foreign key (sid*)* **references** *Sailors,*
foreign key (bid*)* **references** *Boats,*
)

CHECK Constraints:

CHECK constraints allow users to prohibit an operation on a table that would violate theconstraint. It isa local constraint.

Example: To ensure that *rating* must be an integer in the range 1 to 10, we could use:

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10 ))
```

In sailors table if we are trying to insert a new record as
INSERT INTO Sailors
VALUES (5, “Bhupi”, 15, 27.4);
We get insertion is rejected message since value of rating attribute violated the check condition.

Assertions:

Table constraints are associated with a single table, although the conditional expression in the CHECK clause can refer to other tables. Table constraints are required to hold only if the associated table is nonempty. Thus, when a constraint involves two or more tables, the table constraint mechanism is sometimes cumbersome and not quite what is desired. To cover such situations, SQL supports the creation of **assertions**, which are constraints, not associated with any one table.

Assertion in the SQL takes the form

CREATE ASSERTION<assertion_name>CHECK<predicate>
Example, suppose that we wish to enforce the constraint that the number of boats plus the number of sailors should be less than 100.

```
CREATE ASSERTION SailorCheck
CHECK ((SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) <100);
```

Trigger:

A **trigger** is a procedure (statement) that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an **active database**. To design a trigger mechanism, we must meet following three requirements:

1. **Event**: A change to the database that **activates** the trigger.
2. **Condition**: A query or test that is run when the trigger is activated.
3. **Action**: A procedure that is executed when the trigger is activated and its condition is true.

Need for Triggers:

Triggers are useful mechanisms for alerting humans or for starting certain tasks automatically when certain conditions are met. As an illustration, suppose that, instead of allowing negative account balances, the bank deals with overdrafts by setting the account balance to zero, and creating a loan in the amount of the overdraft. The bank gives this loan a loan number identical to the account number of the overdrawn account. For this example, the condition for executing the trigger is an update to the *account* relation that results in a negative *balance* value. Suppose that Jones' withdrawal of some money from an account made the account balance negative. Let *t* denote the account tuple with a negative *balance* value. The actions to be taken are:

Unit 6
Relational Database Design

1. Introduction to anomalies
2. Functional dependencies
3. Decomposition
4. Introduction to Normalization
   - ✓ First Normal Form
   - ✓ Second Normal Form
   - ✓ Third Normal Form
   - ✓ BCNF

While designing a database out of an entity-relationship model, the main problem existing in that raw database is redundancy. Redundancy is storing the same data item in more than one place. A redundancy creates several problems like the following:-

- ✓ Extra storage space: storing the same data in many places takes large amount of disk space.
- ✓ Entering same data more than once during data insertion
- ✓ Deleting data from more than one place during deletion
- ✓ Modifying data in more than one place
- ✓ Anomalies may occur in the database if insertion, deletion, modification etc are not done properly. It creates inconsistency and unreliability in the database.

To solve this problem, the raw database needs to be normalized. This is step by step process of removing different kinds of redundancy and anomaly at each step. At each step a specific rule is followed to remove specific kind of impurity in order to give the database a slim and clean look.

Note:- The process of reducing data redundancy in a relational database is called normalization.

Introduction to Anomalies:-
An anomaly is simply an error or inconsistency in the database. An anomaly may occur in the database if insertion, deletion, modification etc are not done properly. It creates inconsistency and unreliability in the database. There are three types of anomalies may occur in relational database.

- ✓ Insertion Anomalies: Occurs during the insertion of a record. It is the inablility to add data to the database due to absence of other data.
- ✓ Deletion Anomalies: Occurs during the deletion of record. It is unintended loss of data due to deletion of other data.
- ✓ Update Anomalies: Occurs during the updating of a record. It is unintended modification of related data due to updating of a particular data.

Functional Dependencies :-

Functional dependency plays a key role in differentiating good database design from bad database design. It describes the relationship between attributes(columns) in a table.
Let X and Y are attributes of a relation R. If each value of Y is associated with exactly one value of X then Y is said to be functionally dependent on X. It is denoted by X->Y.
So the functional dependency X->Y holds if whenever two tuples have the same value for X, they must have the same value for Y . For any tuples t1 and t2 in any relation instance r(R):
if t1[X]=t2[X], then t1[Y]=t2[Y]

Customer(Cid,Cname,Age,Salary)

Cid->Cname
It is true because Cid uniquely determine the customer name even in the case of duplicate Cname.
Cname-> Cid
It is false because the name of the customer may be same for different Cid. So Cname not uniquely determine the customer.
cid-> Age [True]
cid-> salary[True]
Age-> cid [False]

Example 2:-Consider a relation supplier
Supplier(<u>supplier_id</u>,sname,status, city)
Here, sname,status and city are functionally dependent on supplier_id. Meaning is that each supplier_id uniquely determines the value of attributes supplier_name, supplier_status and city. This can be expressed by
supplier_id->sname
supplier_id-> status
supplier_id-> city

Types of Functional Dependency:-

There are many types of functional dependencies, depending on several criteria.

Fully functional dependency

For a relation schema R, in a functional dependency X->Y, Y is said to be fully functional dependent on X.

Partial Functional Dependency

For a relation schema R, in a functional dependency X->Y, Y is said to be partial functional dependent on X if by removal of some attributes from X and the dependency still holds.

Example

The dependency {Emp_id,Proj_No} -> emp_name is partial becauase only emp_id-> emp_name can holds.

Transitive Dependency

if X->Y and Y->Z then X->Z

Trivial and Non-Trivial dependencies:-

A functional dependency X->Y is said to be trivial dependency. If Y is subset of X(not necessarily a proper subset)

The functional dependency that satisfied by all the relations is called trivial dependency.

Example A->A is satisfied by all the relations involving attribute A .

The functional dependency that is not trivial is said to be non-trivial dependency.

Armstrong Axioms (Inference Rules for FDs) :-

- **Re**flexive:- If Y subset of X, then X->Y
- Augmentation: If X-> Y , then XZ->YZ
- Transitive: If X->Y and Y->Z , then X->Z
- Decomposition: If X->YZ then X-> Y and X->Z
- Union: If X->Y and X->Z then X->YZ
- Pseudo-transitivity: If X->Y and WY->Z, then WX->Z

Closure of Set of Functional Dependencies:-

The **Closure** Of **Functional Dependency** means the complete **set** of all possible attributes that can be functionally derived from given **functional dependency** using the inference rules known as Armstrong's Rules. If “F” is a **functional dependency** then **closure** of **functional dependency** can be denoted using “$\{F\}^+$”.

Closure Of Functional Dependency : Introduction

The Closure Of Functional Dependency means the complete set of all possible attributes that can be functionally derived from given functional dependency using the inference rules known as Armstrong’s Rules.

If “F” is a functional dependency then closure of functional dependency can be denoted using “$\{F\}^+$”.

There are three steps to calculate closure of functional dependency. These are:

Step-1 : Add the attributes which are present on Left Hand Side in the original functional dependency.

Step-2 : Now, add the attributes present on the Right Hand Side of the functional dependency.

Step-3 : With the help of attributes present on Right Hand Side, check the other attributes that can be derived from the other given functional dependencies. Repeat this process until all the possible attributes which can be derived are added in the closure.

Seems difficult? Check out the example explained below and it will surely clear your doubt on how to calculate closure of functional dependency.

Closure Of Functional Dependency : Examples

Example-1 : Consider the table student_details having (Roll_No, Name,Marks, Location) as the attributes and having two functional dependencies.

FD1 : Roll_No-> Name, Marks

FD2 : Name -> Marks, Location

Now, We will calculate the closure of all the attributes present in the relation using the three steps mentioned below.

Step-1 : Add attributes present on the LHS of the first functional dependency to the closure.

$\{Roll\_no\}^+ = \{Roll\_No\}$

Step-2 : Add attributes present on the RHS of the original functional dependency to the closure.

$\{Roll\_no\}^+ = \{Roll\_No, Marks\}$

Step-3 : Add the other possible attributes which can be derived using attributes present on the RHS of the closure. So Roll_No attribute cannot functionally determine any attribute but Name attribute can determine other attributes such as Marks and Location using 2^{nd} Functional Dependency(Name Marks, Location).

Therefore, complete closure of Roll_No will be :

$\{Roll\_no\}^+ = \{Roll\_No, Marks, Name, Location\}$

Similarly, we can calculate closure for other attributes too i.e “Name”.

Step-1 : Add attributes present on the LHS of the functional dependency to the closure.

$\{Name\}^+ = \{Name\}$

Step-2 : Add the attributes present on the RHS of the functional dependency to the closure.

$\{Name\}^+ = \{Name, Marks, Location\}$

Step-3 : Since, we don’t have any functional dependency where “Marks or Location” attribute is functionally determining any other attribute , we cannot add more attributes to the closure. Hence complete closure of Name would be :

$\{Name\}^+ = \{Name, Marks, Location\}$

NOTE : We don’t have any Functional dependency where marks and location can functionally determine any attribute. Hence, for those attributes we can only add the attributes themselves in their closures. Therefore,

$\{Marks\}^+ = \{Marks\}$

and

$\{Location\}^+ = \{ Location\}$

Example-2 : Consider a relation R(A,B,C,D,E) having below mentioned functional dependencies.

FD1 : A -> BC

FD2 : C -> B

FD3 : D -> E

FD4 : E -> D

Now, we need to calculate the closure of attributes of the relation R. The closures will be:

$\{A\}^+ = \{A, B, C\}$

$\{B\}^+ = \{B\}$

$\{C\}^+ = \{B, C\}$

$\{D\}^+ = \{D, E\}$

$\{E\}^+ = \{E\}$

Closure Of Functional Dependency : Calculating Candidate Key

"A Candidate Key of a relation is an attribute or set of attributes that can determine the whole relation or contains all the attributes in its closure."

Let's try to understand how to calculate candidate keys.

Example-1 : Consider the relation R(A,B,C) with given functional dependencies :

FD1 : A -> B

FD2 : B -> C

Now, calculating the closure of the attributes as :

$\{A\}^+ = \{A, B, C\}$

$\{B\}^+ = \{B, C\}$

$\{C\}^+ = \{C\}$

Clearly, "A" is the candidate key as, its closure contains all the attributes present in the relation "R".

Example-2 : Consider another relation R(A, B, C, D, E) having the Functional dependencies :

FD1 : A -> BC

FD2 : C -> B

FD3 : D -> E

FD4 : E -> D

Now, calculating the closure of the attributes as :

$\{A\}^+ = \{A, B, C\}$

$\{B\}^+ = \{B\}$

$\{C\}^+ = \{C, B\}$

$\{D\}^+ = \{E, D\}$

$\{E\}^+ = \{E, D\}$

In this case, a single attribute does is unable to determine all the attribute on its own like in previous example. Here, we need to club two or more attributes to determine the candidate keys.

$\{A, D\}^+ = \{A, B, C, D, E\}$

$\{A, E\}^+ = \{A, B, C, D, E\}$

Hence, "AD" and "AE" are the two possible keys of the given relation “R”. Any other combination other than these two would have acted as extraneous attributes.

NOTE : Any relation “R” can have either single or multiple candidate keys.

Closure Of Functional Dependency : Key Definitions

Prime Attributes : Attributes which are indispensable part of candidate keys. For example : “A, D, E” attributes are prime attributes in above example-2.

Non-Prime Attributes : Attributes other than prime attributes which does not take part in formation of candidate keys. For example.

Extraneous Attributes : Attributes which does not make any effect on removal from candidate key.

For example : Consider the relation R(A, B, C, D) with functional dependencies :

FD1 : A -> BC

FD2 : B -> C

FD3 : D -> C

Here, Candidate key can be “AD” only. Hence,

Prime Attributes : A, D.

Non-Prime Attributes : B, C

Extraneous Attributes : B, C(As if we add any of the to the candidate key, it will remain unaffected).

Those attributes, which if removed does not affect closure of that set.

Example 1: Let R = (A,B,C,D)

F={A->B,A->C,BC->D}. List several member of F^+.

Soln:-

Since A->B,A->C then A->BC [By Union Rule]

Since A->BC , BC->D then AB-> D [By pseudo transitivity rule]

Hence F^+ ={A->B,A->C,BC->D,A->BC,A-->D,AB->D}

Example 2: let R=(A,B,C,G,H,I)
F={A->B, A->C,CG->H,CG->I,B->H} .Compute closure of F (F^+)

Sol^n:-
since A->B,B->H then A->H [By transitivity rule]
since A->B,A->C then A->BC [by union rule]
since A->C then AG->CG [by augmentation rule]
since AG->CG, CG-> I then AG->I [by transitivity rule]
since CG->H , CG->I then CG-> HI [by union rule]
Augmentation of CG->I to infer CG->CGI , augmentation of CG->H to infer CGI->HI and then transitivity.
Hence, F+= {A->A ,B->B,C->C,H->H,G->G,I->I,A->B,A->C,CG->H,G->I,CG->HI,B->H,A->H,AG->I,CG->HI}
Here, first six FDs obtain by reflexive axioms.

Example 3: let R=(A,B,C,D) and F={A->B,A->C,BC->D} then compute F^+

since A->B and A->C then by union rule A->BC
Since BC->D then by projective/Decomposition B->D,C->D Again by transitivity A->B and B->D and A->C and C->D => A->D
Hence F^+ = { A->A,B->B,C->C , D->D,A->B,A->C,BC->D,B->D,C->D,A->D}

Normalization:-

1 **Informal Design Guidelines for Relational Databases:-** Semantics of the Relation Attributes,Redundant Information in Tuples and Update Anomalies, Null Values in Tuples , Spurious Tuples

2 **Functional Dependencies (FDs):** Definition of FD, Inference Rules for FDs, Equivalence of Sets of FDs, Minimal Sets of FDs

3 **Normal Forms Based on Primary Keys:- N**ormalization of Relations , Practical Use of Normal Forms , Definitions of Keys and Attributes Participating in Keys , First Normal Form, Second Normal Form, Third Normal Form

4 General Normal Form Definitions (For Multiple Keys)

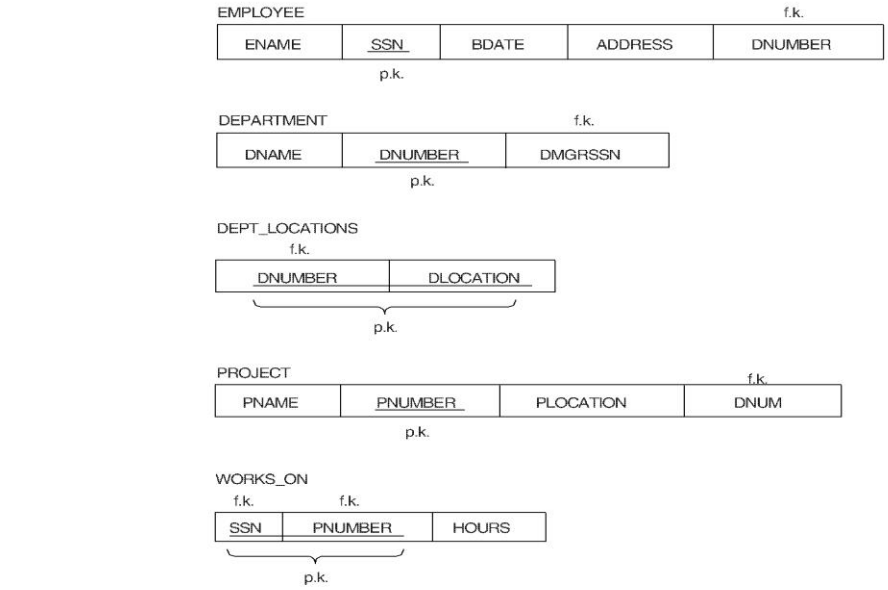
5 BCNF (Boyce-Codd Normal Form)

GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).

- Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
- Only foreign keys should be used to refer to other entities
- Entity and relationship attributes should be kept apart as much as possible.

Bottom Line: Design a schema that can be explained easily relation by relation. The semantics of attributes should be easy to interpret.

Figure 14.1 Simplified version of the COMPANY relational database schema.



1.2 Redundant Information in Tuples and Update Anomalies

- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
  - Problems with update anomalies: Insertion anomalies, Deletion anomalies , Modification anomalies

EXAMPLE OF AN UPDATE ANOMALY

Consider the relation: EMP_PROJ (Emp#, Proj#, Ename, Pname, No_hours)

Update Anomaly: Changing the name of project number P1 from "Billing" to "Customer-Accounting" may cause this update to be made for all 100 employees working on project P1.

- **Insert Anomaly:** Cannot insert a project unless an employee is assigned to .

  Inversely - Cannot insert an employee unless an he/she is assigned to a project.

- **Delete Anomaly:** When a project is deleted, it will result in deleting all the employees who work on that project. Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Figure 14.3 Two relation schemas and their functional dependencies. Both suffer from update anomalies. (a) The EMP_DEPT relation schema. (b) The EMP_PROJ relation schema.

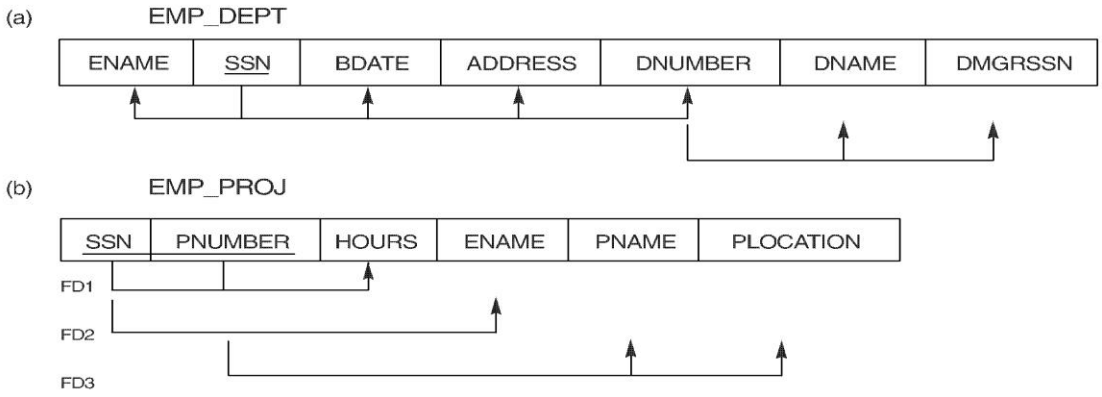


Figure 14.4 Example relations for the schemas in Figure 14.3 that result from applying NATURAL JOIN to the relations in Figure 14.2. These may be stored as base relations for performance reasons.

EMP_DEPT

ENAME	SSN	BDATE	ADDRESS	DNUMBER	DNAME	DMGRSSN
Smith,John B.	123456789	1965-01-09	731 Fondren,Houston,TX	5	Research	333445555
Wong,Franklin T.	333445555	1955-12-08	638 Voss,Houston,TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle,Spring,TX	4	Administration	987654321
Wallace,Jennifer S.	987654321	1941-06-20	291 Berry,Bellaire,TX	4	Administration	987654321
Narayan,Ramesh K.	666884444	1962-09-15	975 FireOak,Humble,TX	5	Research	333445555
English,Joyce A.	453453453	1972-07-31	5631 Rice,Houston,TX	5	Research	333445555
Jabbar,Ahmad V.	987987987	1969-03-29	980 Dallas,Houston,TX	4	Administration	987654321
Borg,James E.	888665555	1937-11-10	450 Stone,Houston,TX	1	Headquarters	888665555

EMP_PROJ

SSN	PNUMBER	HOURS	ENAME	PNAME	PLOCATION
123456789	1	32.5	Smith,John B.	ProductX	Bellaire
123456789	2	7.5	Smith,John B.	ProductY	Sugarland
666884444	3	40.0	Narayan,Ramesh K.	ProductZ	Houston
453453453	1	20.0	English,Joyce A.	ProductX	Bellaire
453453453	2	20.0	English,Joyce A.	ProductY	Sugarland
333445555	2	10.0	Wong,Franklin T.	ProductY	Sugarland
333445555	3	10.0	Wong,Franklin T.	ProductZ	Houston
333445555	10	10.0	Wong,Franklin T.	Computerization	Stafford
333445555	20	10.0	Wong,Franklin T.	Reorganization	Houston
999887777	30	30.0	Zelaya,Alicia J.	Newbenefits	Stafford
999887777	10	10.0	Zelaya,Alicia J.	Computerization	Stafford
987987987	10	35.0	Jabbar,Ahmad V.	Computerization	Stafford
987987987	30	5.0	Jabbar,Ahmad V.	Newbenefits	Stafford
987654321	30	20.0	Wallace,Jennifer S.	Newbenefits	Stafford
987654321	20	15.0	Wallace,Jennifer S.	Reorganization	Houston
888665555	20	null	Borg,James E.	Reorganization	Houston

- **GUIDELINE 2:** Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then note them so that applications can be made to take them into account

GUIDELINE 3: Relations should be designed such that their tuples will have as few NULL values as possible

- Attributes that are NULL frequently could be placed in separate relations (with the primary key)
- Reasons for nulls:
  - attribute not applicable or invalid , attribute value unknown (may exist) ,value known to exist, but unavailable

1.4 Spurious Tuples

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations

GUIDELINE 4: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.

There are two important properties of decompositions:

(a) non-additive or losslessness of the corresponding join

(b) preservation of the functional dependencies.

Note that property (a) is extremely important and *cannot* be sacrificed. Property (b) is less stringent and may be sacrificed. (See Chapter 11).

2.1 Functional Dependencies

- Functional dependencies (FDs) are used to specify *formal measures* of the "goodness" of relational designs
- FDs and keys are used to define **normal forms** for relations
- FDs are **constraints** that are derived from the *meaning* and *interrelationships* of the data attributes
- A set of attributes X *functionally determines* a set of attributes Y if the value of X determines a unique value for Y
- X -> Y holds if whenever two tuples have the same value for X, they *must have* the same value for Y
- For any two tuples t1 and t2 in any relation instance r(R): *If* t1[X]=t2[X], *then* t1[Y]=t2[Y]
- X -> Y in R specifies a *constraint* on all relation instances r(R)
- Written as X -> Y; can be displayed graphically on a relation schema as in Figures. (denoted by the arrow:).
- FDs are derived from the real-world constraints on the attributes

Examples of FD constraints

- social security number determines employee name , SSN -> ENAME
- project number determines project name and location , PNUMBER -> {PNAME, PLOCATION}
- employee ssn and project number determines the hours per week that the employee works on the project. {SSN, PNUMBER} -> HOURS

Examples of FD constraints

- An FD is a property of the attributes in the schema R
- The constraint must hold on *every relation instance* r(R)
- If K is a key of R, then K functionally determines all attributes in R (since we never have two distinct tuples with t1[K]=t2[K])

2.2 Inference Rules for FDs

- Given a set of FDs F, we can *infer* additional FDs that hold whenever the FDs in F hold

Armstrong's inference rules:

IR1. (**Reflexive**) If Y *subset-of* X, then X -> Y

IR2. (**Augmentation**) If X -> Y, then XZ -> YZ , (Notation: XZ stands for X U Z)

IR3. (**Transitive**) If X -> Y and Y -> Z, then X -> Z

- IR1, IR2, IR3 form a *sound* and *complete* set of inference rules

Some **additional inference rules** that are useful:

(**Decomposition**) If X -> YZ, then X -> Y and X -> Z

(**Union**) If X -> Y and X -> Z, then X -> YZ

(**Psuedotransitivity**) If X -> Y and WY -> Z, then WX -> Z

- The last three inference rules, as well as any other inference rules, can be deduced from IR1, IR2, and IR3 (completeness property)
- **Closure** of a set F of FDs is the set F^+ of all FDs that can be inferred from F
- **Closure** of a set of attributes X with respect to F is the set X^+ of all attributes that are functionally determined by X
- X^+ can be calculated by repeatedly applying IR1, IR2, IR3 using the FDs in F

Equivalence of Sets of FDs

- Two sets of FDs F and G are **equivalent** if:

  - every FD in F can be inferred from G, *and* , - every FD in G can be inferred from F
- Hence, F and G are equivalent if $F^+ = G^+$

Definition: F **covers** G if every FD in G can be inferred from F (i.e., if G^+ *subset-of* F^+)

- F and G are equivalent if F covers G and G covers F
- There is an algorithm for checking equivalence of sets of FDs

2.4 Minimal Sets of FDs

- A set of FDs is **minimal** if it satisfies the following conditions:

(1) Every dependency in F has a single attribute for its RHS.

(2) We cannot remove any dependency from F and have a set of dependencies that is equivalent to F.

(3) We cannot replace any dependency X -> A in F with a dependency Y -> A, where Y proper-subset-of X (Y subset-of X) and still have a set of dependencies that is equivalent to F.

(4) Every set of FDs has an equivalent minimal set, There can be several equivalent minimal sets

(5) There is no simple algorithm for computing a minimal set of FDs that is equivalent to a set F of FDs

(6) To synthesize a set of relations, we assume that we start with a set of dependencies that is a minimal set (e.g., see algorithms 11.2 and 11.4)

Normal Forms Based on Primary Keys

The normalization process, as first proposed by Codd (1972a), takes a relation schema through a series of tests to *certify* whether it satisfies a certain **normal form**. The process, which proceeds in a top-down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary, can thus be considered as *relational design by analysis.* Initially, Codd proposed three normal forms, which he called first, second, and third normal form. A stronger def-inition of 3NF—called Boyce-Codd normal form (BCNF)—was proposed later by Boyce and Codd. All these normal forms are based on a single analytical tool: the functional dependencies among the attributes of a relation. Later, a fourth normal form (4NF) and a fifth normal form (5NF) were proposed, based on the concepts of multivalued dependencies and join dependencies.

Normalization of data can be considered a process of analyzing the given relation schemas based on their FDs and primary keys to achieve the desirable properties of minimizing redundancy and (2) minimizing the insertion, deletion, and update anomalies. It can be considered as a "filtering" or "purifi-cation" process to make the design have successively better quality. Unsatisfactory relation schemas that do not meet certain conditions—the **normal form tests**—are decomposed into smaller relation schemas that meet the tests and hence possess the desirable properties. Thus, the normalization procedure provides database design-ers with the following:

A formal framework for analyzing relation schemas based on their keys and on the functional dependencies among their attributes

A series of normal form tests that can be carried out on individual relation schemas so that the relational database can be **normalized** to any desired degree

Definition. The **normal form** of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been nor-malized.

- **Normalization**: The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

- **Normal form**: Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form

- 2NF, 3NF, BCNF based on keys and FDs of a relation schema
- 4NF based on keys, multi-valued dependencies : MVDs; 5NF based on keys, join dependencies : JDs (Chapter 11)
- Additional properties may be needed to ensure a good relational design (lossless join, dependency preservation; Chapter 11)
- **Normalization** is carried out in practice so that the resulting designs are of high quality and meet the desirable properties
- The practical utility of these normal forms becomes questionable when the constraints on which they are based are **hard to understand** or to **detect**
- The database designers ***need not*** normalize to the highest possible normal form. (usually up to 3NF, BCNF or 4NF)
- **Denormalization:** the process of storing the join of higher normal form relations as a base relation—which is in a lower normal form
- A **superkey** of a relation schema $R = \{A_1, A_2, ..., A_n\}$ is a set of attributes S *subset-of* R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **key** K is a superkey with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more.
- If a relation schema has more than one key, each is called a **candidate key.** One of the candidate keys is *arbitrarily* designated to be the **primary key,** and the others are called *secondary keys*.
- A **Prime attribute** must be a member of *some candidate key*
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

3.2 First Normal Form

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; historically, it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF dis-allows *relations within relations* or *relations as attribute values within tuples*. The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values**.

- Disallows composite attributes, multivalued attributes, and **nested relations**; attributes whose values *for an individual tuple* are non-atomic
- Considered to be part of the definition of relation

Figure 14.8 Normalization into 1NF. (a) Relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Figure 14.9 Normalizing nested relations into 1NF. (a) Schema of the EMP_PROJ relation with a "nested relation" PROJS. (b) Example extension of the EMP_PROJ relation showing nested relations within each tuple. (c) Decomposing EMP_PROJ into 1NF relations EMP_PROJ1 and EMP_PROJ2 by propagating the primary key.

(a) **EMP_PROJ**

SSN	ENAME	PROJS	
		PNUMBER	HOURS

(b) **EMP_PROJ**

SSN	ENAME	PNUMBER	HOURS
123456789	Smith,John B.	1	32.5
		2	7.5
666884444	Narayan,Ramesh K.	3	40.0
453453453	English,Joyce A.	1	20.0
		2	20.0
333445555	Wong,Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya,Alicia J.	30	30.0
		10	10.0
987987987	Jabbar,Ahmad V.	10	35.0
		30	5.0
987654321	Wallace,Jennifer S.	30	20.0
		20	15.0
888665555	Borg,James E.	20	null

(c) **EMP_PROJ1**

SSN	ENAME

EMP_PROJ2

SSN	PNUMBER	HOURS

3.3 Second Normal Form

- Uses the concepts of **FD**s, **primary key**

- **Second normal form (2NF)** is based on the concept of *full functional dependency*. A functional dependency $X \rightarrow Y$ is a **full functional dependency** if removal of any attribute A from X means that the dependency does not hold any more; that is, for any attribute $A \in X$, $(X - \{A\})$ does *not* functionally determine Y. A functional dependency $X \rightarrow Y$ is a **partial dependency** if some attribute $A \in X$ can be removed from X and the dependency still holds; that is, for some $A \in X$, $(X - \{A\}) \rightarrow Y$. In Figure 15.3(b), {Ssn, Pnumber} → Hours is a full dependency (neither Ssn → Hours nor Pnumber → Hours holds). However, the dependency {Ssn, Pnumber} → Ename is partial because Ssn → Ename holds.
-
- **Definition.** A relation schema R is in 2NF if every nonprime attribute A in R is *fully functionally dependent* on the primary key of R.

Definitions:

- **Prime attribute** - attribute that is member of the primary key K

- **Full functional dependency** - a FD Y -> Z where removal of any attribute from Y means the FD does not hold any more

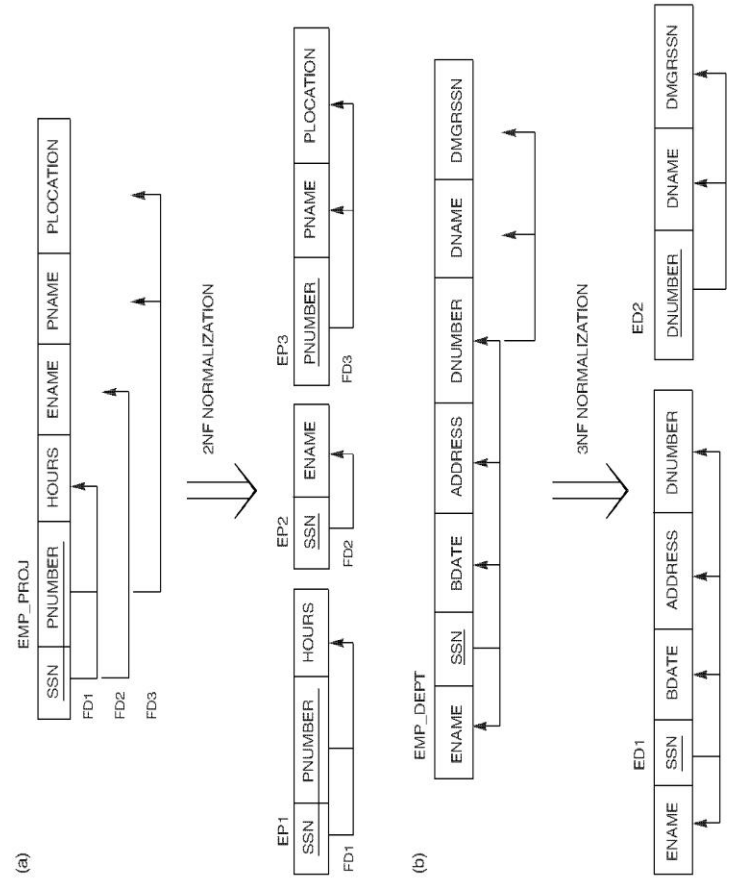
  Examples: - {SSN, PNUMBER} -> HOURS is a full FD since neither SSN -> HOURS nor PNUMBER -> HOURS hold

  - {SSN, PNUMBER} -> ENAME is *not* a full FD (it is called a *partial dependency*) since SSN -> ENAME also holds

- A relation schema R is in **second normal form** (**2NF**) if every non-prime attribute A in R is fully functionally dependent on the primary key

- R can be decomposed into 2NF relations via the process of 2NF normalization

Figure 14.10 The normalization process. (a) Normalizing EMP_PROJ into 2NF relations. (b) Normalizing EMP_DEPT into 3NF relations.



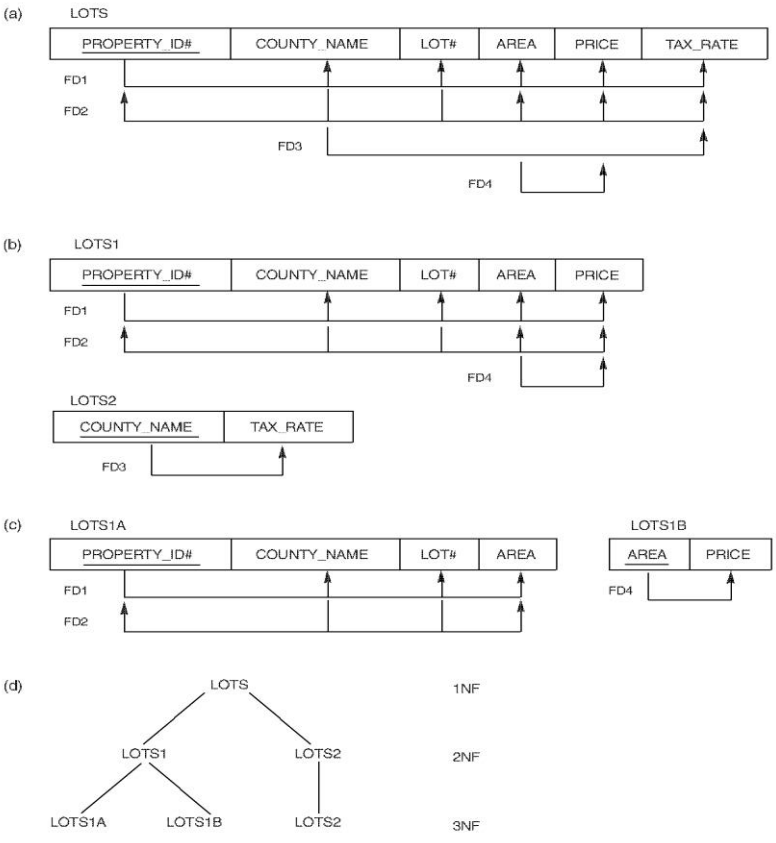


Figure 14.11 Normalization to 2NF and 3NF. (a) The lots relation schema and its functional dependencies fd1 through FD4. (b) Decomposing lots into the 2NF relations LOTS1 and LOTS2. (c) Decomposing LOTS1 into the 3NF relations LOTS1A and LOTS1B. (d) Summary of normalization of lots.

Third Normal Form

Definition:

- **Transitive functional dependency** - a FD X -> Z that can be derived from two FDs X -> Y and Y -> Z

  Examples:

  - SSN -> DMGRSSN is a *transitive* FD since

  SSN -> DNUMBER and DNUMBER -> DMGRSSN hold

- SSN -> ENAME is *non-transitive* since there is no set of attributes X where SSN -> X and X -> ENAME

Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency $X \rightarrow Y$ in a relation schema R is a **transitive dependency** if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key of R,[10] and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. The dependency Ssn $\rightarrow$ Dmgr_ssn is transitive through Dnumber in EMP_DEPT

- A relation schema R is in **third normal form** (**3NF**) if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key
- R can be decomposed into 3NF relations via the process of 3NF normalization

NOTE:

In X -> Y and Y -> Z, with X as the primary key, we consider this a problem only if Y is <u>not</u> a candidate key. When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).

Here, SSN -> Emp# -> Salary and Emp# is a candidate key.

General Normal Form Definitions

<u>Definition:</u>

- Superkey of relation schema R - a set of attributes S of R that contains a key of R
- A relation schema R is in third normal form (3NF) if whenever a FD X -> A holds in R, then either:

(a) X is a superkey of R, or , (b) A is a prime attribute of R

NOTE: Boyce-Codd normal form disallows condition (b) above

5 BCNF (Boyce-Codd Normal Form)

- A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X -> A holds in R, then X is a superkey of R
- Each normal form is strictly stronger than the previous one
  - Every 2NF relation is in 1NF, Every 3NF relation is in 2NF, Every BCNF relation is in 3NF

- There exist relations that are in 3NF but not in BCNF , The goal is to have each relation in BCNF (or 3NF)

Figure 14.12 Boyce-Codd normal form. (a) BCNF normalization with the dependency of FD2 being "lost" in the decomposition. (b) A relation *R* in 3NF but not in BCNF.

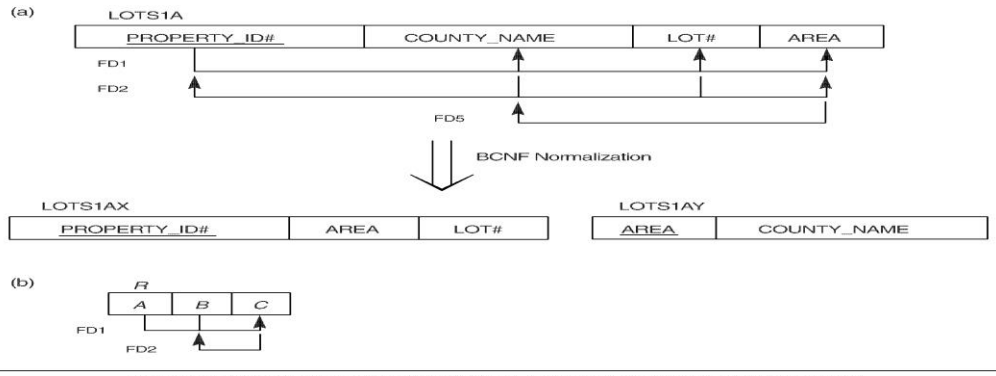


Figure 14.13 A relation TEACH that is in 3NF but not in BCNF.

TEACH

STUDENT	COURSE	INSTRUCTOR
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe

Achieving the BCNF by Decomposition

- Two FDs exist in the relation TEACH: fd1: { student, course} -> instructor

  fd2: instructor -> course

- {student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 10.12 (b). So this relation is in 3NF <u>but not in</u> BCNF

- A relation NOT in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations. (See Algorithm 11.3)

Achieving the BCNF by Decomposition

- Three possible decompositions for relation TEACH

  1. {<u>student, instructor</u>} and {<u>student, course</u>}
  2. {course, <u>instructor</u> } and {<u>course, student</u>}

3. {instructor, course } and {instructor, student}

- All three decompositions will lose fd1. We have to settle for sacrificing the functional dependency preservation. But we cannot sacrifice the non-additivity property after decomposition.
- Out of the above three, only the 3rd decomposition will not generate spurious tuples after join.(and hence has the non-additivity property).
- A test to determine whether a binary decomposition (decomposition into two relations) is nonadditive (lossless) is discussed in section 11.1.4 under Property LJ1. Verify that the third decomposition above meets the property.

Transaction concurrency control Recovery

1. Introduction to Transaction Processing Concepts and Theory

The concept of transaction provides a mechanism for describing logical units of database processing.
Transaction processing systems are systems with large databases and hundreds of concurrent users executing database transactions. Examples of such systems include airline reservations, banking, credit card processing, online retail purchasing, stock markets, supermarket checkouts, and many other applications. These systems require high availability and fast response time for hundreds of concurrent users. We define the concept of a transaction, which is used to represent a logical unit of database processing that must be completed in its entirety to ensure correct-ness. A transaction is typically implemented by a computer program, which includes database commands such as retrievals, insertions, deletions, and updates.

Introduction to Transaction Processing

1.1 Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system **concurrently**. A DBMS is **single-user** if at most one user at a time can use the system, and it is **multiuser** if many users can use the system—and hence access the database—concurrently. Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, an airline reservations system is used by hundreds of travel agents and reservation clerks concurrently. Database systems used in banks, insurance agencies, stock exchanges, supermarkets, and many other applications are multiuser systems. In these systems, hundreds or thousands of users are typically operating on the data-base by submitting transactions concurrently to the system process, and so on. A process is resumed at the point where it was suspended when-ever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually **interleaved**, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), **parallel process-ing** of multiple processes is possible, as illustrated by processes C and D in Figure. Most of the theory concerning concurrency control in databases is developed in terms of **interleaved concurrency.** In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

1.1.2 Transactions, Database Items, Read and Write Operations, and DBMS Buffers

A **transaction** is an executing program that forms a logical unit of database process-ing. A transaction includes one or more database access operations—these can include insertion, deletion, modification, or

retrieval operations. The database operations that form a transaction can either be embedded within an application program or they can be specified interactively via a high-level query language such as SQL. One way of specifying the transaction boundaries is by specifying explicit **begin transaction** and **end transaction** statements in an application program; in this case, all database access operations between the two are considered as forming one transaction. A single application program may contain more than one transac-tion if it contains several transaction boundaries. If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**; otherwise it is known as a **read-write transaction**.

The *database model* that is used to present transaction processing concepts is quite simple when compared to the data models that we discussed earlier in the book, such as the relational model or the object model. A **database** is basically represented as a collection of *named data items.* The size of a data item is called its **granularity**. A **data item** can be a *database record*, but it can also be a larger unit such as a whole *disk block*, or even a smaller unit such as an individual *field (attribute) value* of some record in the database. The transaction processing concepts we discuss are inde-pendent of the data item granularity (size) and apply to data items in general. Each data item has a *unique name*, but this name is not typically used by the programmer; rather, it is just a means to *uniquely identify each data item*. For example, if the data item granularity is one disk block, then the disk block address can be used as the data item name. Using this simplified database model, the basic database access operations that a transaction can include are as follows:

read_item(*X*). Reads a database item named *X* into a program variable. To simplify our notation, we assume that *the program variable is also named X*.

write_item(*X*). Writes the value of program variable *X* into the database item named *X*.

The basic unit of data transfer from disk to main memory is one block.

Executing a read_item(*X*) command includes the following steps:

Find the address of the disk block that contains item *X*. Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

Copy item *X* from the buffer to the program variable named *X*.

Executing a write_item(*X*) command includes the following steps:

Find the address of the disk block that contains item *X*.

Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).

Copy item *X* from the program variable named *X* into its correct location in the buffer.

Store the updated block from the buffer back to disk (either immediately or at some later point in time).

It is step 4 that actually updates the database on disk. In some cases the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer. Usually, the decision about when to store a modified disk block whose contents are in a main memory buffer is handled by the recovery manager of the DBMS in coop-eration with the underlying operating system. The DBMS will maintain in the **database cache** a number of **data buffers** in main memory. Each buffer typically holds the contents of one database disk block, which contains some of the database items being processed. When these buffers are all occupied, and additional database disk blocks must be copied into memory, some buffer

replacement policy is used to choose which of the current buffers is to be replaced. If the chosen buffer has been modified, it must be written back to disk before it is reused.

A transaction includes read_item and write_item operations to access and update the database. Figure 1.2 shows examples of two very simple transactions. The **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that the transaction writes. For example, the read-set of T_1 in Figure 1.2 is {*X*, *Y*} and its write-set is also {*X*, *Y*}.

Concurrency control and recovery mechanisms are mainly concerned with the database commands in a transaction. Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is *uncontrolled*, it may lead to problems, such as an incon-sistent database. In the next section we informally introduce some of the problems that may occur.

1.1.3 Why Concurrency Control Is Needed

Several problems can occur when concurrent transactions execute in an uncon-trolled manner. We illustrate some of these problems by referring to a much simpli-fied airline reservations database in which a record is stored for each airline flight. Each record includes the *number of reserved seats* on that flight as a *named (uniquely identifiable) data item*, among other information. Figure 1.2(a) shows a transac-tion T_1 that *transfers N* reservations from one flight whose number of reserved seats is stored in the database item named *X* to another flight whose number of reserved seats is stored in the database item named *Y*. Figure 1.2(b) shows a simpler trans-action T_2 that just *reserves M* seats on the first flight (*X*) referenced in transaction T_1.[2] To simplify our example, we do not show additional portions of the transac-tions, such as checking whether a flight has enough seats available before reserving additional seats.

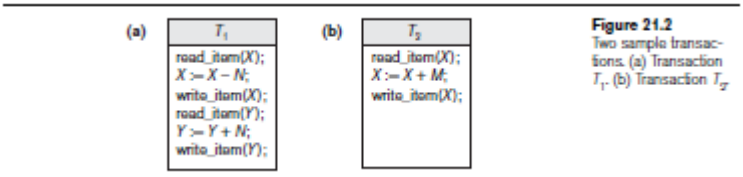


Figure 21.2
Two sample transac-tions. (a) Transaction T_1. (b) Transaction T_2.

Figure 1.2

Two sample transac-tions. (a) Transaction T_1. (b) Transaction T_2.

We will not discuss buffer replacement policies here because they are typically discussed in operating systems textbooks.

A similar, more commonly used example assumes a bank database, with one transaction doing a trans-fer of funds from account *X* to account *Y* and the other transaction doing a deposit to account *X*.

When a database access program is written, it has the flight number, flight date, and the number of seats to be booked as parameters; hence, the same program can be used to execute *many*

different transactions, each with a different flight number, date, and number of seats to be booked. For concurrency control purposes, a trans-action is a *particular execution* of a program on a specific date, flight, and number of seats. In Figure 1.2(a) and (b), the transactions T_1 and T_2 are *specific executions* of the programs that refer to the specific flights whose numbers of seats are stored in data items X and Y in the database. Next we discuss the types of problems we may encounter with these two simple transactions if they run concurrently.

The Lost Update Problem. This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect. Suppose that transactions T_1 and T_2 are submitted at approximately the same time, and suppose that their operations are interleaved as shown in Figure 21.3(a); then the final value of item X is incorrect because T_2 reads the value of X *before* T_1 changes it in the database, and hence the updated value resulting from T_1 is lost. For example, if $X = 80$ at the start (originally there were 80 reservations on the flight), $N = 5$ (T_1 transfers 5 seat reservations from the flight corresponding to X to the flight corresponding to Y), and $M = 4$ (T_2 reserves 4 seats on X), the final result should be $X = 79$. However, in the interleaving of operations shown in Figure 21.3(a), it is $X = 84$ because the update in T_1 that removed the five seats from X was *lost*.

The Temporary Update (or Dirty Read) Problem. This problem occurs when one transaction updates a database item and then the transaction fails for some rea-son. Meanwhile, the updated item is accessed (read) by another transaction before it is changed back to its original value. Figure 1.3(b) shows an example where T_1 updates item X and then fails before completion, so the system must change X back to its original value. Before it can do so, however, transaction T_2 reads the *temporary* value of X, which will not be recorded permanently in the data-base because of the failure of T_1. The value of item X that is read by T_2 is called *dirty data* because it has been created by a transaction that has not completed and com-mitted yet; hence, this problem is also known as the *dirty read problem*.

The Incorrect Summary Problem. If one transaction is calculating an aggregate summary function on a number of database items while other transactions are updating some of these items, the aggregate function may calculate some values before they are updated and others after they are updated. For example, suppose that a transaction T_3 is calculating the total number of reservations on all the flights; meanwhile, transaction T_1 is executing. If the interleaving of operations shown in Figure 21.3(c) occurs, the result of T_3 will be off by an amount N because T_3 reads the value of X *after* N seats have been

subtracted from it but reads the value of *Y before* those *N* seats have been added to it.

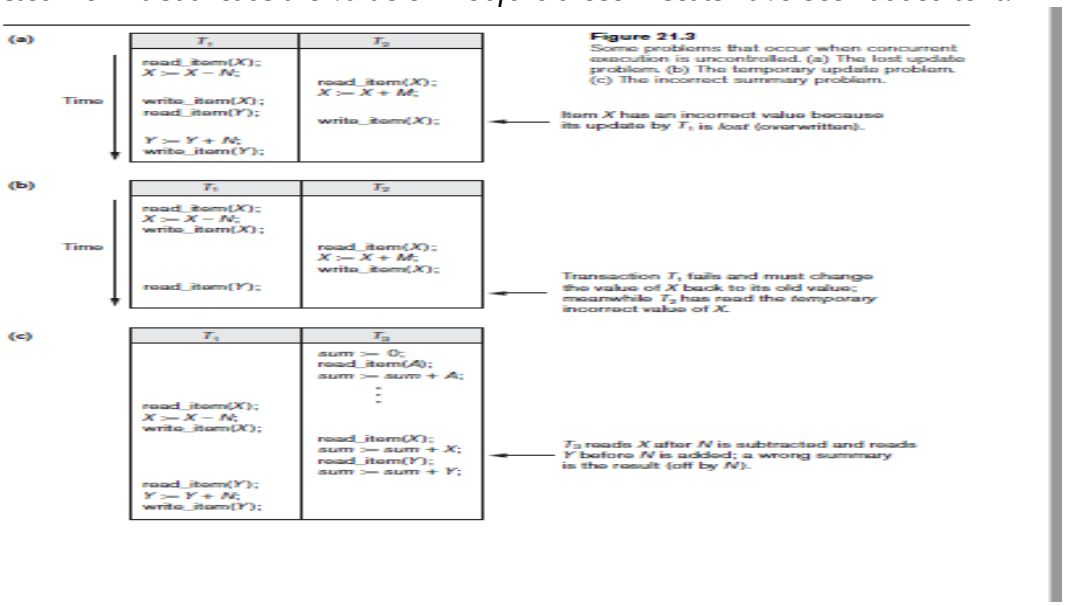


Figure 1.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.
Item *X* has an incorrect value because its update by T_1 is *lost* (overwritten).

Transaction T_1 fails and must change the value of *X* back to its old value; meanwhile T_2 has read the *temporary* incorrect value of *X*.

T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

The Unrepeatable Read Problem. Another problem that may occur is called *unrepeatable read*, where a transaction *T* reads the same item twice and the item is changed by another transaction *T* between the two reads. Hence, *T* receives *different values* for its two reads of the same item. This may occur, for example, if during an airline reservation transaction, a customer inquires about seat availability on several flights. When the customer decides on a particular flight, the transaction then reads the number of seats on that flight a second time before completing the reservation, and it may end up reading a different value for the item.

Concurrency Control :-

==========

•Process of managing simultaneous operations on the database without having them interfere with one another.

• Prevents interference when two or more users are accessing database simultaneously and at least one

is updating data.

• Although two transactions may be correct in themselves, interleaving of operations may produce an incorrect result.

Need for Concurrency Control

Three examples of potential problems caused by concurrency:

- Lost update problem
- Uncommitted dependency problem
- Inconsistent analysis problem.

Lost Update Problem

Successfully completed update is overridden by another user.

Example:

- T_1 withdraws £10 from an account with bal_x, initially £100.
- T_2 deposits £100 into same account.
- Serially, final balance would be £190.

Lost Update Problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	$bal_x = bal_x + 100$	100
t_4	$bal_x = bal_x - 10$	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

Loss of T_2's update!!

This can be avoided by preventing T_1 from reading bal_x until after update.

Uncommitted Dependency Problem

Occurs when one transaction can see intermediate results of another transaction before it has committed.

Example:

- T_4 updates bal_x to £200 but it aborts, so bal_x should be back at original value of £100.
- T_3 has read new value of bal_x (£200) and uses value as basis of £10 reduction, giving a new balance of £190, instead of £90.

Uncommitted Dependency Problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		$bal_x = bal_x + 100$	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	!	200
t_6	$bal_x = bal_x - 10$	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

Problem avoided by preventing T3 from reading bal_x until after T4 commits or aborts.

Inconsistent Analysis Problem

Occurs when transaction reads several values but second transaction updates some of them during execution of first.

Example:

- T6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meantime, T5 has transferred £10 from bal_x to bal_z, so T6 now has wrong result (£10 too high).

Why Recovery Is Needed

Whenever a transaction is submitted to a DBMS for execution, the system is respon-sible for making sure that either all the operations in the transaction are completed successfully and their effect is recorded permanently in the database, or that the transaction does not have any effect on the database or any other transactions. In the first case, the transaction is said to be **committed**, whereas in the second case, the transaction is **aborted**. The DBMS must not permit some operations of a trans-action *T* to be applied to the database while other operations of *T* are not, because *the whole transaction* is a logical unit of database processing. If a transaction **fails** after executing some of its operations but before executing all of them, the opera-tions already executed must be undone and have no lasting effect.

Types of Failures. Failures are generally classified as transaction, system, and media failures. There are several possible reasons for a transaction to fail in the mid-dle of execution:

A computer failure (system crash). A hardware, software, or network error occurs in the computer system during transaction execution. Hardware crashes are usually media failures—for example, main memory failure.

A transaction or system error. Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may

also occur because of erroneous parameter values or because of a logical programming error.[3] Additionally, the user may interrupt the transaction during its execution.

Local errors or exception conditions detected by the transaction. During transaction execution, certain conditions may occur that necessitate cancel-lation of the transaction. For example, data for the transaction may not be found. An exception condition,[4] such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal, to be canceled. This exception could be programmed in the transaction itself, and in such a case would not be considered as a transaction failure.

Disk failure. Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

Physical problems and catastrophes. This refers to an endless list of prob-lems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Failures of types 1, 2, 3, and 4 are more common than those of types 5 or 6. Whenever a failure of type 1 through 4 occurs, the system must keep sufficient information to quickly recover from the failure. Disk failure or other catastrophic failures of type 5 or 6 do not happen frequently; if they do occur, recovery is a major task. we will discuss recovery in later.

The concept of transaction is fundamental to many techniques for concurrency control and recovery from failures.

1.2 Transaction and System Concepts

A transaction is an atomic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits or aborts.Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

BEGIN_TRANSACTION. This marks the beginning of transaction execution.

READ or WRITE. These specify read or write operations on the database items that are executed as part of a transaction.

END_TRANSACTION. This specifies that READ and WRITE transaction oper-ations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability.

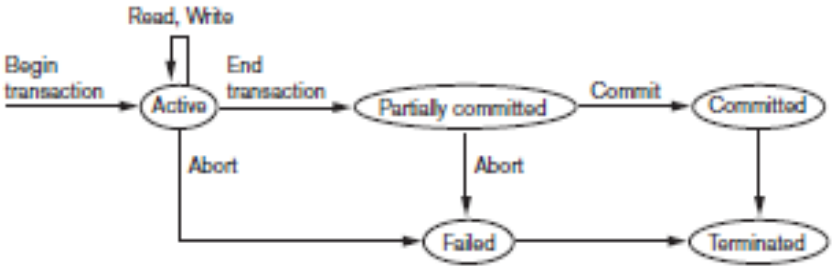
COMMIT_TRANSACTION. This signals a *successful end* of the transaction so that any changes (updates) executed by the transaction can be safely **committed** to the database and will not be undone.

ROLLBACK (or ABORT). This signals that the transaction has *ended unsuc-cessfully*, so that any changes or effects that the transaction may have applied to the database must be **undone**.

Figure 1.4 shows a state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an **active state** immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the **partially committed state**. At this point, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log, discussed in the next section). Once this check is successful, the transaction is said to have reached its commit point and enters the **committed state**. When a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database, even if a system failure occurs.

However, a transaction can go to the **failed state** if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The **terminated state** corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been run-ning is removed when the transaction terminates. Failed or aborted transactions may be *restarted* later—either automatically or after being resubmitted by the user—as brand new transactions.

Figure 21.4
State transition diagram illustrating the states for transaction execution.



[5]Optimistic concurrency control (see Section 22.4) also requires that certain checks are made at this point to ensure that the transaction did not interfere with other executing transactions.

1.2.2 The System Log

To be able to recover from failures that affect transactions, the system maintains a **log** to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recov-ery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one (or more) main memory buffers hold the last part of the log file, so that log entries are first added to the main memory buffer. When the **log buffer** is filled, or when certain other conditions occur, the log buffer is *appended to the end of the log file on disk*. In addition, the log file from disk is periodically backed up to archival storage (tape) to guard against catastrophic failures. The following are the types of entries—called **log records**—that are written to the log file and the corresponding action for each log record. In these entries, *T* refers to a unique **transaction-id** that is generated automatically by the system for each transaction and that is used to identify each transaction:

[start_transaction**,** ***T*****].** Indicates that transaction *T* has started execution.

[write_item**,** ***T, X, old_value, new_value*****]**. Indicates that transaction *T* has changed the value of database item *X* from *old_value* to *new_value.*

[read_item**,** ***T, X*****]**. Indicates that transaction *T* has read the value of database item *X*.

[commit**,** ***T*****]**. Indicates that transaction *T* has completed successfully, and affirms that its effect can be committed (recorded permanently) to the data-base.

[abort**,** ***T*****]**. Indicates that transaction *T* has been aborted.

Protocols for recovery that avoid cascading rollbacks which include nearly all practical protocols—*do not require* that READ operations are writ-ten to the system log. However, if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included. Additionally, some recovery protocols require simpler WRITE entries only include one of new_value and old_value instead of including both

Notice that we are assuming that all permanent changes to the database occur within transactions, so the notion of recovery from a transaction failure amounts to either undoing or redoing transaction operations individually from the log. If the system crashes, we can recover to a consistent database state. Because the log contains a record of every WRITE operation that changes the value of some database item, it is possible to **undo** the effect of these WRITE operations of a transaction *T* by tracing backward through the log and resetting all items changed by a WRITE operation of *T* to their old_values. **Redo** of an operation may also be necessary if a transaction has its updates recorded in the log but a failure occurs before the system can be sure that The log has sometimes been called the *DBMS journal*. all these new_values have been written to the actual database on disk from the main memory buffers.[7]

1.2.3 Commit Point of a Transaction

A transaction *T* reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction opera-tions on the database have been recorded in the log. Beyond the commit point, the transaction is said to be **committed**, and its effect must be *permanently recorded* in the database. The transaction then writes a commit record [commit, *T*] into the log. If a system failure occurs, we can search back in the log for all transactions *T* that have written a [start_transaction, *T*] record into the log but have not written their [commit, *T*] record yet; these transactions may have to be *rolled back* to *undo their effect* on the database during the recovery process. Transactions that have written their commit record in the log must also have recorded all their WRITE operations in the log, so their effect on the database can be *redone* from the log records.

Notice that the log file must be kept on disk. As discussed in Chapter 17, updating a disk file involves copying the appropriate block of the file from disk to a buffer in main memory, updating the buffer in main memory, and copying the buffer to disk. It is common to keep one or more blocks of the log file in main memory buffers, called the **log buffer**, until they are filled with log entries and then to write them back to disk only once, rather than writing to disk every time a log entry is added. This saves the overhead of multiple disk writes of the same log file buffer. At the time of a system crash, only the log entries that have been *written back to disk* are considered in the recovery process because the contents of main memory may be lost. Hence, *before* a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk. This process is called **force-writing** the log buffer before committing a transaction.

1.3 Desirable Properties of Transactions (ACID) Properties

Transactions should possess several properties, often called the **ACID** properties; they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

Atomicity. A transaction is an atomic unit of processing; it should either be performed in its entirety or not performed at all.

Consistency preservation. A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

Isolation. A transaction should appear as though it is being executed in iso-lation from other transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

Durability or permanency. The changes applied to the database by a com-mitted transaction must persist in the database. These changes must not be lost because of any failure.

The *atomicity property* requires that we execute a transaction to completion. It is the responsibility of the *transaction recovery subsystem* of a DBMS to ensure atomicity. If a transaction fails to complete for some reason, such as a system crash in the midst of transaction execution, the recovery technique must undo any effects of the transaction on the database. On the other hand, write operations of a committed transaction must be eventually written to disk.

The preservation of *consistency* is generally considered to be the responsibility of the programmers who write the database programs or of the DBMS module that enforces integrity constraints. Recall that a **database state** is a collection of all the stored data items (values) in the database at a given point in time. A **consistent state** of the database satisfies the constraints specified in the schema as well as any other constraints on the database that should hold. A database program should be written in a way that guarantees that, if the database is in a consistent state before executing the transaction, it will be in a consistent state after the *complete* execution of the transaction, assuming that *no interference with other transactions* occurs.

The *isolation property* is enforced by the *concurrency control subsystem* of the DBMS. If every transaction does not make its updates (write operations) visible to other transactions until it is committed, one form of isolation is enforced that solves the temporary update problem and eliminates cascading rollbacks but does not eliminate all other problems. There have been attempts to define the **level of isolation** of a transaction. A transaction is said to have level 0 (zero) isola-tion if it does not overwrite the dirty reads of higher-level transactions. Level 1 (one) isolation has no lost updates, and level 2 isolation has no lost updates and no dirty reads. Finally, level 3 isolation (also called *true isolation*) has, in addition to level 2 properties, repeatable reads.[9]

And last, the *durability property* is the responsibility of the *recovery subsystem* of the DBMS.

1.4 Characterizing Schedules Based on Recoverability

When transactions are executing concurrently in an interleaved fashion, then the order of execution of operations from all the various transactions is known as a **schedule** (or **history**). In this section, first we define the concept of schedules, and then we characterize the types of schedules that facilitate recovery when failures occur. we characterize schedules in terms of the interference of par-ticipating transactions, leading to the concepts of serializability and serializable schedules.

1.4.1 Schedules (Histories) of Transactions

A **schedule** (or **history**) S of n transactions T_1, T_2, ..., T_n is an ordering of the oper-ations of the transactions. Operations from different transactions can be interleaved in the schedule S. However, for each

transaction T_i that participates in the schedule S, the operations of T_i in S must appear in the same order in which they occur in T_i. The order of operations in S is considered to be a *total ordering*, meaning *that for any two operations* in the schedule, one must occur before the other. It is possible theoretically to deal with schedules whose operations form *partial orders* (as we discuss later), but we will assume for now total ordering of the operations in a schedule.

For the purpose of recovery and concurrency control, we are mainly interested in the read_item and write_item operations of the transactions, as well as the commit and abort operations. A shorthand notation for describing a schedule uses the symbols *b*, *r*, *w*, *e*, *c*, and *a* for the operations begin_transaction, read_item, write_item, end_transac-tion, commit, and abort, respectively, and appends as a *subscript* the transaction id (transaction number) to each operation in the schedule. In this notation, the data-base item X that is read or written follows the *r* and *w* operations in parentheses. In some schedules, we will only show the *read* and *write* operations, whereas in other schedules, we will show all the operations. For example, the schedule in Figure 21.3(a), which we shall call S_a, can be written as follows in this notation:

S_a: $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$;

Similarly, the schedule for Figure 21.3(b), which we call S_b, can be written as follows, if we assume that transaction T_1 aborted after its read_item(Y) operation:

S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1;

Two operations in a schedule are said to **conflict** if they satisfy all three of the fol-lowing conditions: (1) they belong to *different transactions*; (2) they access the *same item X*; and (3) *at least one* of the operations is a write_item(X). For example, in schedule S_a, the operations $r_1(X)$ and $w_2(X)$ conflict, as do the operations $r_2(X)$ and $w_1(X)$, and the operations $w_1(X)$ and $w_2(X)$. However, the operations $r_1(X)$ and $r_2(X)$ do not conflict, since they are both read operations; the operations $w_2(X)$ and $w_1(Y)$ do not conflict because they operate on distinct data items X and Y; and the operations $r_1(X)$ and $w_1(X)$ do not conflict because they belong to the same transaction.

Intuitively, two operations are conflicting if changing their order can result in a dif-ferent outcome. For example, if we change the order of the two operations $r_1(X)$; $w_2(X)$ to $w_2(X)$; $r_1(X)$, then the value of X that is read by transaction T_1 changes, because in the second order the value of X is changed by $w_2(X)$ before it is read by $r_1(X)$, whereas in the first order the value is read before it is changed. This is called a **read-write conflict**. The other type is called a **write-write conflict**, and is illustrated by the case where we change the order of two operations such as $w_1(X)$; $w_2(X)$ to $w_2(X)$; $w_1(X)$. For a write-write conflict, the *last value* of X will differ because in one case it is written by T_2 and in the other case by T_1. Notice that two read operations are not conflicting because changing their order makes no difference in outcome.

The rest of this section covers some theoretical definitions concerning schedules. A schedule S of n transactions T_1, T_2, ..., T_n is said to be a **complete schedule** if the following conditions hold:

The operations in S are exactly those operations in T_1, T_2, ..., T_n, including a commit or abort operation as the last operation for each transaction in the schedule.

For any pair of operations from the same transaction T_i, their relative order of appearance in S is the same as their order of appearance in T_i.

For any two conflicting operations, one of the two must occur before the other in the schedule.

The preceding condition (3) allows for two *nonconflicting operations* to occur in the schedule without defining which occurs first, thus leading to the definition of a schedule as a **partial order** of the operations in the n transactions.[11] However, a total order must be specified in the schedule for any pair of conflicting operations (condition 3) and for any pair of operations from the same transaction (condition 2). Condition 1 simply states that all operations in the transactions must appear in the complete schedule. Since every transaction has either committed or aborted, a complete schedule will *not contain any active transactions* at the end of the schedule.

In general, it is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system. Hence, it is useful to define the concept of the **committed projection** $C(S)$ of a schedule S, which includes only the operations in S that belong to committed trans-actions—that is, transactions T_i whose commit operation c_i is in S.

1.4.2 Characterizing Schedules Based on Recoverability

For some schedules it is easy to recover from transaction and system failures, whereas for other schedules the recovery process can be quite involved. In some cases, it is even not possible to recover correctly after a failure. Hence, it is important to characterize the types of schedules for which *recovery is possible*, as well as those for which *recovery is relatively simple*. These characterizations do not actually pro-vide the recovery algorithm; they only attempt to theoretically characterize the dif-ferent types of schedules.

First, we would like to ensure that, once a transaction T is committed, it should *never* be necessary to roll back T. This ensures that the durability property of trans-actions is not violated (see Section 21.3). The schedules that theoretically meet this criterion are called *recoverable schedules;* those that do not are called **nonrecoverable** and hence should not be permitted by the DBMS. The definition of **recoverable schedule** is as follows: A schedule S is recoverable if no transaction T in S commits until all transactions T that have written some item X that T reads have committed. A transaction T **reads** from transaction T in a schedule S if some item X is first written by T and later read by T. In addition, T should not have been aborted before T reads item X, and there should be no transactions that write X after T writes it and before T reads it (unless those transactions, if any, have aborted before T reads X).

Some recoverable schedules may require a complex recovery process as we shall see, but if sufficient information is kept (in the log), a recovery algorithm can be devised for any recoverable schedule. The (partial) schedules S_a and S_b from the preceding section are both recoverable, since they satisfy the above definition. Consider the schedule S_a given below, which is the same as schedule S_a except that two commit operations have been added to S_a:

S_a : $r_1(X)$; $r_2(X)$; $w_1(X)$; $r_1(Y)$; $w_2(X)$; c_2; $w_1(Y)$; c_1;

S_a is recoverable, even though it suffers from the lost update problem; this problem is handled by serializability theory (see Section 21.5). However, consider the two (partial) schedules S_c and S_d that follow:

S_c: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; c_2; a_1;
S_d: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; c_1; c_2;
S_e: $r_1(X)$; $w_1(X)$; $r_2(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$; a_1; a_2;

S_c is not recoverable because T_2 reads item X from T_1, but T_2 commits before T_1 commits. The problem occurs if T_1 aborts after the c_2 operation in S_c, then the value of X that T_2 read is no longer valid and T_2 must be aborted *after* it is committed, leading to a schedule that is *not recoverable*. For the schedule to be recoverable, the c_2 operation in S_c must be postponed until after T_1 commits, as shown in S_d. If T_1 aborts instead of committing, then T_2 should also abort as shown in S_e, because the value of X it read is no longer valid. In S_e, aborting T_2 is acceptable since it has not committed yet, which is not the case for the nonrecoverable schedule S_c.

In a recoverable schedule, no committed transaction ever needs to be rolled back, and so the definition of committed transaction as durable is not violated. However, it is possible for a phenomenon known as **cascading rollback** (or **cascading abort**) to occur in some recoverable schedules, where an *uncommitted* transaction has to be rolled back because it read an item from a transaction that failed. This is illustrated in schedule S_e, where transaction T_2 has to be rolled back because it read item X from T_1, and T_1 then aborted.

Because cascading rollback can be quite time-consuming—since numerous transac-tions can be rolled back—it is important to characterize the sched- ules where this phenomenon is guaranteed not to occur. A schedule is said to be **cascadeless**, or to **avoid cascading rollback**, if every transaction in the schedule reads only items that were written by committed transactions. In this case, all items read will not be discarded, so no cascading rollback will occur. To satisfy this criterion, the $r_2(X)$ command in schedules S_d and S_e must be postponed until after T_1 has commit-ted (or aborted), thus delaying T_2 but ensuring no cascading rollback if T_1 aborts.

Finally, there is a third, more restrictive type of schedule, called a **strict schedule**, in which transactions can *neither read nor write* an item X until the last transaction that wrote X has committed (or aborted). Strict schedules simplify the recovery process. In a strict schedule, the process of undoing a write_item(X) operation of an aborted transaction is simply to restore the **before image** (old_value or BFIM) of data item X. This simple procedure always works correctly for strict schedules, but it may not work for recoverable or cascadeless schedules. For example, consider schedule S_f :

S_f : $w_1(X, 5)$; $w_2(X, 8)$; a_1;

Suppose that the value of X was originally 9, which is the before image stored in the system log along with the $w_1(X, 5)$ operation. If T_1 aborts, as in S_f , the recovery pro-cedure that restores the before image of an aborted write operation will restore the value of X to 9, even though it has already been changed to 8 by transaction T_2, thus leading to potentially incorrect results. Although schedule S_f is cascadeless, it is not a strict schedule, since it permits T_2 to write item X even though the transaction T_1 that last wrote X had not yet committed (or aborted). A strict schedule does not have this problem.

It is important to note that any strict schedule is also cascadeless, and any cascade-less schedule is also recoverable. Suppose we have i transactions T_1, T_2, ..., T_i, and their number of operations are n_1, n_2, ..., n_i, respectively. If we make a set of all pos-sible schedules of these transactions, we can divide the schedules into two disjoint subsets: recoverable and nonrecoverable. The cascadeless schedules will be a subset of the recoverable schedules, and the strict schedules will be a subset of the cascade-less schedules. Thus, all strict schedules are cascadeless, and all cascadeless schedules are recoverable.

1.5 Characterizing Schedules Based on Serializability

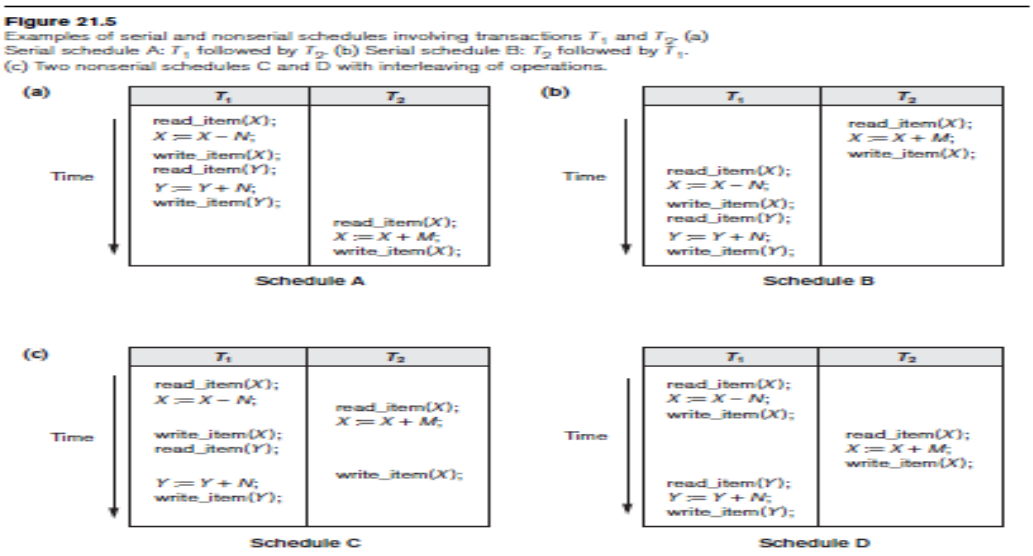
In the previous section, we characterized schedules based on their recoverability properties. Now we characterize the types of schedules that are always considered to be *correct* when concurrent transactions are executing. Such schedules are known as *serializable schedules*. Suppose that two users—for example, two airline reservations agents—submit to the DBMS transactions T_1 and T_2 in Figure 21.2 at approxi-mately the same time. If no interleaving of operations is permitted, there are only two possible outcomes:

Execute all the operations of transaction T_1 (in sequence) followed by all the operations of transaction T_2 (in sequence). Execute all the operations of transaction T_2 (in sequence) followed by all the operations of transaction T_1 (in sequence).

These two schedules—called *serial schedules*—are shown in Figure 21.5(a) and (b), respectively. If interleaving of operations is allowed, there will be many possible orders in which the system can execute the individual operations of the transac-tions. Two possible schedules are shown in Figure 21.5(c). The concept of **serializability of schedules** is used to identify which schedules are correct when transaction executions have interleaving of their operations in the schedules. This section defines serializability and discusses how it may be used in practice.

Figure 1.5

Examples of serial and nonserial schedules involving transactions T_1 and T_2. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1. (c) Two nonserial schedules C and D with interleaving of operations.



1.5.1 Serial, Nonserial, and Conflict-Serializable Schedules

Schedules A and B in Figure 1.5(a) and (b) are called *serial* because the operations of each transaction are executed consecutively, without any interleaved operations from the other transaction. In a serial schedule, entire transactions are performed in serial order: T_1 and then T_2 in Figure 1.5(a), and T_2 and then T_1 in Figure 1.5(b). Schedules C and D in Figure 1.5(c) are called *nonserial* because each sequence interleaves operations from the two transactions.

Formally, a schedule *S* is **serial** if, for every transaction *T* participating in the sched-ule, all the operations of *T* are executed consecutively in the schedule; otherwise, the schedule is called **nonserial**. Therefore, in a serial schedule, only one transaction at a time is active—the commit (or abort) of the active transaction initiates execution of the next transaction. No interleaving occurs in a serial schedule. One reasonable assumption we can make, if we consider the transactions to be *independent*, is that *every serial schedule is considered correct*. We can assume this because every transac-tion is assumed to be correct if executed on its own (according to the *consistency preservation* property of Section 21.3). Hence, it does not matter which transaction is executed first. As long as every transaction is executed from beginning to end in isolation from the operations of other transactions, we get a correct end result on the database.

The problem with serial schedules is that they limit concurrency by prohibiting interleaving of operations. In a serial schedule, if a transaction waits for an I/O operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time. Additionally, if some transaction *T* is quite long, the other transactions must wait for *T* to complete all its operations before starting. Hence, serial schedules are *considered unacceptable* in practice. However, if we can determine which other schedules are *equivalent* to a serial sched-ule, we can allow these schedules to occur.

To illustrate our discussion, consider the schedules in Figure 1.5, and assume that the initial values of database items are $X = 90$ and $Y = 90$ and that $N = 3$ and $M = 2$. After executing transactions T_1 and T_2, we would expect the database values to be *X*

89 and $Y = 93$, according to the meaning of the transactions. Sure enough, execut-ing either of the serial schedules A or B gives the correct results. Now consider the nonserial schedules C and D. Schedule C (which is the same as Figure 1.3(a)) gives the results $X = 92$ and $Y = 93$, in which the *X* value is erroneous, whereas schedule D gives the correct results.

Schedule C gives an erroneous result because of the *lost update problem* discussed in Section 21.1.3; transaction T_2 reads the value of *X* before it is changed by transac-tion T_1, so only the effect of T_2 on *X* is reflected in the database. The effect of T_1 on *X* is *lost*, overwritten by T_2, leading to the incorrect result for item *X*. However, some nonserial schedules give the correct expected result, such as schedule D. We would like to determine which of the nonserial schedules *always* give a correct result and which may give erroneous results. The concept used to characterize schedules in this manner is that of serializability of a schedule.

The definition of *serializable schedule* is as follows: A schedule *S* of *n* transactions is **serializable** if it is *equivalent to some serial schedule* of the same *n* transactions. We will define the concept of *equivalence of schedules* shortly. Notice that there are *n*! possible serial schedules of *n* transactions and many more possible nonserial sched-ules. We can form two disjoint groups of the nonserial schedules—those that are equivalent to one (or more) of the serial schedules and hence are serializable, and those that are not equivalent to *any* serial schedule and hence are not serializable.

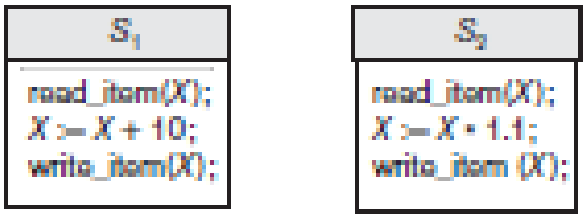
Saying that a nonserial schedule *S* is serializable is equivalent to saying that it is cor-rect, because it is equivalent to a serial schedule, which is considered correct. The remaining question is: When are two schedules considered *equivalent*?

There are several ways to define schedule equivalence. The simplest but least satis-factory definition involves comparing the effects of the schedules on the database. Two schedules are called **result equivalent** if they produce the same final state of the database. However, two different schedules may accidentally produce the same final state. For example, in Figure 21.6, schedules S_1 and S_2 will produce the same final database state if they execute on a database with an initial value of $X = 100$; however, for other initial values of *X*, the schedules are *not* result equivalent. Additionally, these schedules execute different transactions, so they definitely should not be con-sidered equivalent. Hence, result equivalence alone cannot be used to define equiv-alence of schedules. The safest and most general approach to defining schedule equivalence is not to make any assumptions about the types of operations included in the transactions. For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules *in the same order*. Two definitions of equivalence of schedules are gener-ally used: *conflict equivalence* and *view equivalence*. We discuss conflict equivalence next, which is the more commonly used definition.

The definition of *conflict equivalence* of schedules is as follows: Two schedules are said to be **conflict equivalent** if the order of any two *conflicting operations* is the same in both schedules. Recall from Section 21.4.1 that two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and either both are write_item operations or one is a write_item and the other a read_item. If two conflicting operations are applied in *different orders* in two sched-ules, the effect can be different on the database or on the transactions in the sched-ule, and hence the schedules are not conflict equivalent. For example, as we discussed in Section 1.4.1, if a read and write operation occur in the order $r_1(X)$, $w_2(X)$ in schedule S_1, and in the reverse order $w_2(X)$, $r_1(X)$ in schedule S_2, the value read by $r_1(X)$ can be different in the two schedules. Similarly, if two write operations

Figure 1.6

Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.



occur in the order $w_1(X)$, $w_2(X)$ in S_1, and in the reverse order $w_2(X)$, $w_1(X)$ in S_2, the next $r(X)$ operation in the two schedules will read potentially different values; or if these are the last operations writing item X in the schedules, the final value of item X in the database will be different. Using the notion of conflict equivalence, we define a schedule S to be **conflict seri-alizable**[12] if it is (conflict) equivalent to some serial schedule S . In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S . According to this definition, schedule D in Figure 1.5(c) is equivalent to the serial schedule A in Figure 21.5(a). In both schedules, the read_item(X) of T_2 reads the value of X written by T_1, while the other operations read the

database values from the initial database state. Additionally, T_1 is the last transaction to write Y, and T_2 is the last transaction to write X in both schedules. Because A is a serial schedule and schedule D is equivalent to A, D is a serializable schedule. Notice that the operations $r_1(Y)$ and $w_1(Y)$ of schedule D do not conflict with the opera-tions $r_2(X)$ and $w_2(X)$, since they access different data items. Therefore, we can move $r_1(Y)$, $w_1(Y)$ before $r_2(X)$, $w_2(X)$, leading to the equivalent serial schedule T_1, T_2.

Schedule C in Figure 21.5(c) is not equivalent to either of the two possible serial schedules A and B, and hence is *not serializable*. Trying to reorder the operations of schedule C to find an equivalent serial schedule fails because $r_2(X)$ and $w_1(X)$ con-flict, which means that we cannot move $r_2(X)$ down to get the equivalent serial schedule T_1, T_2. Similarly, because $w_1(X)$ and $w_2(X)$ conflict, we cannot move $w_1(X)$ down to get the equivalent serial schedule T_2, T_1.

1.5.2 Testing for Conflict Serializability of a Schedule

There is a simple algorithm for determining whether a particular schedule is con-flict serializable or not. Most concurrency control methods do *not* actually test for serializability. Rather protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable.

Algorithm 1.1 can be used to test a schedule for conflict serializability. The algo-rithm looks at only the read_item and write_item operations in a schedule to construct a **precedence graph** (or **serialization graph**), which is a **directed graph** $G = (N, E)$ that consists of a set of nodes $N = \{T_1, T_2, ..., T_n\}$ and a set of directed edges $E = \{e_1, e_2, ..., e_m\}$. There is one node in the graph for each transaction T_i in the schedule. Each edge e_i in the graph is of the form $(T_j \rightarrow T_k)$, $1 \leq j \leq n$, $1 \leq k \phi n$, where T_j is the **starting node** of e_i and T_k is the **ending node** of e_i. Such an edge from node T_j to We will use *serializable* to mean conflict serializable. node T_k is created by the algorithm if one of the operations in T_j appears in the schedule before some *conflicting operation* in T_k.

Algorithm 1.1. Testing Conflict Serializability of a Schedule S

For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.

For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.

For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.

For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X), create an edge ($T_i \rightarrow T_j$) in the precedence graph.

The schedule S is serializable if and only if the precedence graph has no cycles.

The precedence graph is constructed as described in Algorithm 21.1. If there is a cycle in the precedence graph, schedule S is not (conflict) serializable; if there is no cycle, S is serializable. A **cycle** in a directed graph is a **sequence of edges** $C = ((T_j \rightarrow T_k), (T_k \rightarrow T_{p}), ..., (T_i \rightarrow T_j))$ with the property that the starting node of each edge—except the first edge—is the same as the ending node of the previous edge, and the starting node of the first edge is the same as the ending node of the last edge (the sequence starts and ends at the same node).

In the precedence graph, an edge from T_i to T_j means that transaction T_i must come before transaction T_j in any serial schedule that is equivalent to S, because two con-flicting operations appear in the schedule in that order. If there is no cycle in the precedence graph, we can create an **equivalent serial schedule** S that is equivalent to S, by ordering the transactions that participate in S as follows: Whenever an edge exists in the precedence graph from T_i to T_j, T_i must appear before T_j in the equiva-lent serial schedule S .[13] Notice that the edges ($T_i \rightarrow T_j$) in a precedence graph can optionally be labeled by the name(s) of the data item(s) that led to creating the edge. Figure 1.7 shows such labels on the edges.

In general, several serial schedules can be equivalent to S if the precedence graph for S has no cycle. However, if the precedence graph has a cycle, it is easy to show that we cannot create any equivalent serial schedule, so S is not serializable. The prece-dence graphs created for schedules A to D, respectively, in Figure 1.5 appear in Figure 1.7(a) to (d). The graph for schedule C has a cycle, so it is not serializable. The graph for schedule D has no cycle, so it is serializable, and the equivalent serial schedule is T_1 followed by T_2. The graphs for schedules A and B have no cycles, as expected, because the schedules are serial and hence serializable.

This process of ordering the nodes of an acrylic graph is known as *topological sorting*.

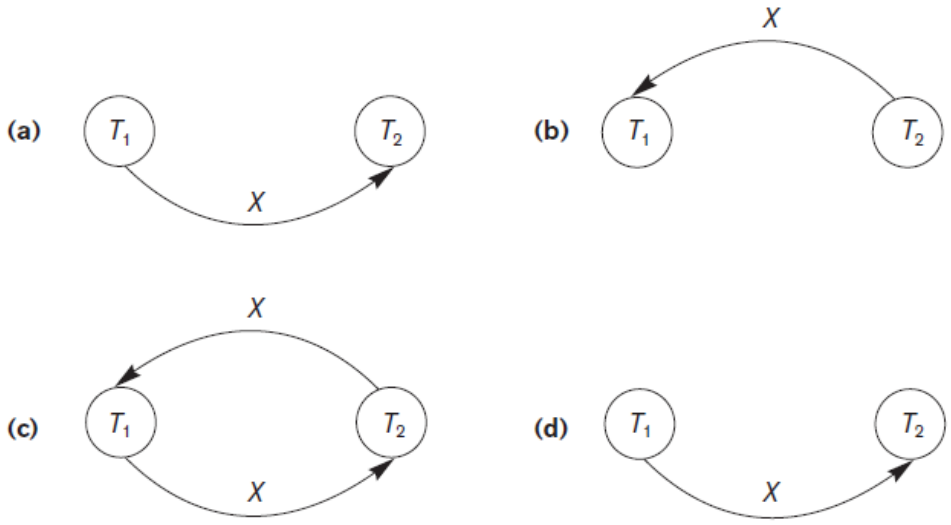


Figure 1.7

Constructing the precedence graphs for schedules A to D from Figure 21.5 to test for conflict serializability. (a) Precedence graph for serial schedule A. (b) Precedence graph for serial schedule B. (c) Precedence graph for schedule C (not serializable). (d) Precedence graph for schedule D (serializable, equivalent to schedule A).

Another example, in which three transactions participate, is shown in Figure 21.8. Figure 1.8(a) shows the read_item and write_item operations in each transaction.

Two schedules *E* and *F* for these transactions are shown in Figure 1.8(b) and (c), respectively, and the precedence graphs for schedules E and F are shown in parts (d) and (e). Schedule E is not serializable because the corresponding precedence graph has cycles. Schedule F is serializable, and the serial schedule equivalent to *F* is shown in Figure 1.8(e). Although only one equivalent serial schedule exists for *F*, in gen-eral there may be more than one equivalent serial schedule for a serializable sched-ule. Figure 1.8(f) shows a precedence graph representing a schedule that has two equivalent serial schedules. To find an equivalent serial schedule, start with a node that does not have any incoming edges, and then make sure that the node order for every edge is not violated.

1.5.3 How Serializability Is Used for Concurrency Control

As we discussed earlier, saying that a schedule *S* is (conflict) serializable—that is, *S* is (conflict) equivalent to a serial schedule—is tantamount to saying that *S* is correct. Being *serializable* is distinct from being *serial*, however. A serial schedule represents inefficient processing because no interleaving of operations from different transac-tions is permitted. This can lead to low CPU utilization while a transaction waits for disk I/O, or for another transaction to terminate, thus slowing down processing considerably. A serializable schedule gives the benefits of concurrent execution without giving up any correctness. In practice, it is quite difficult to test for the seri-alizability of a schedule. The interleaving of operations from concurrent transac-tions—which are usually executed as processes by the operating system—is typically determined by the operating system scheduler, which allocates resources to

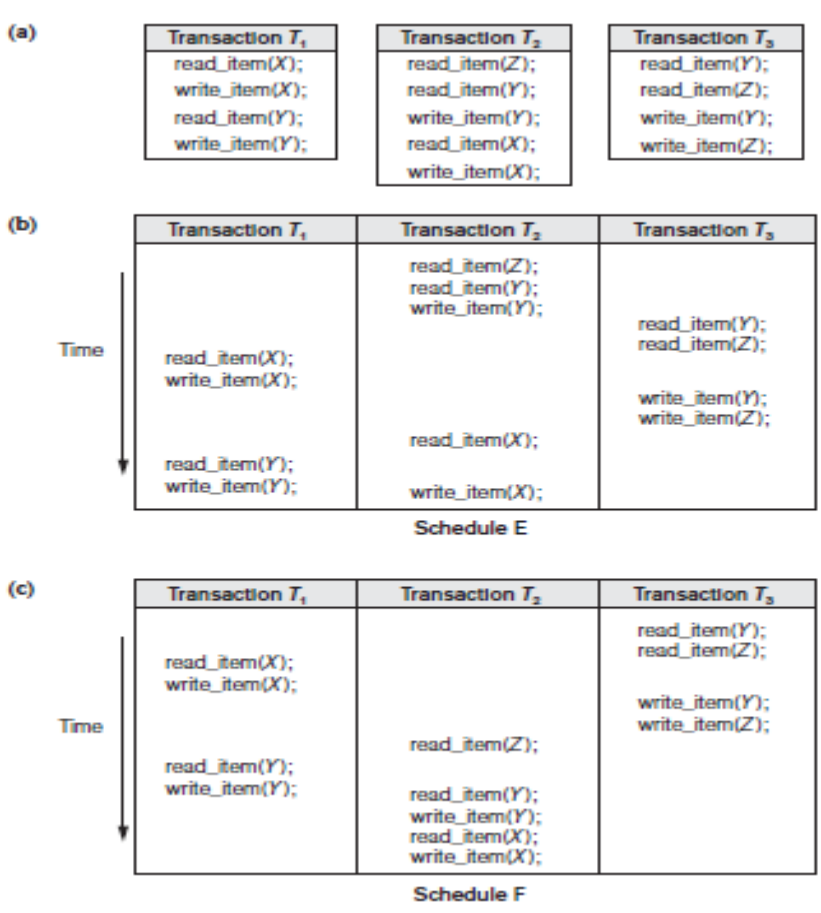


Figure 21.8
Another example of serializability testing. (a) The read and write operations of three transactions T_1, T_2, and T_3. (b) Schedule E. (c) Schedule F.

all processes. Factors such as system load, time of transaction submission, and pri-orities of processes contribute to the ordering of operations in a schedule. Hence, it is difficult to determine how the operations of a schedule will be interleaved before-hand to ensure serializability.

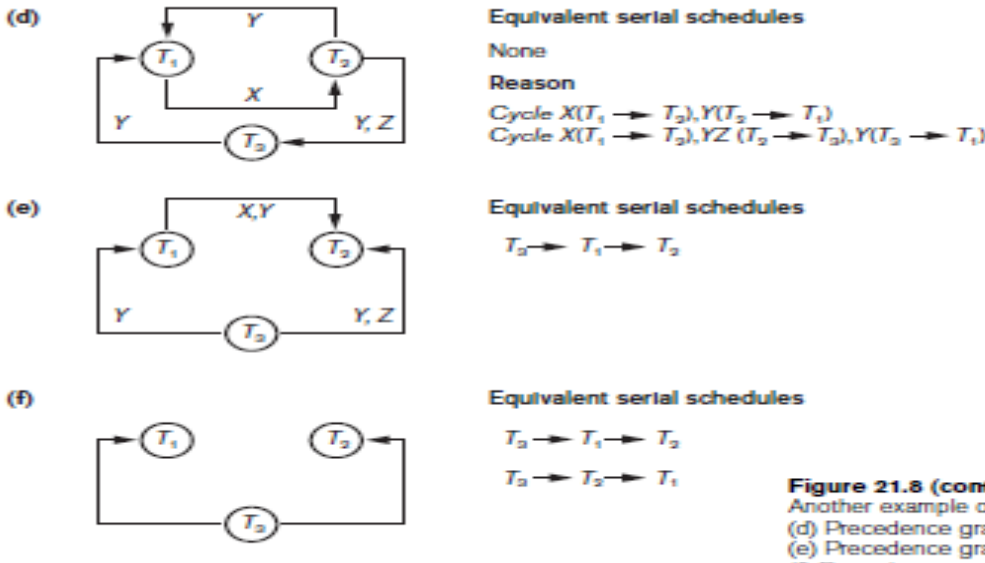


Figure 21.8 (continued)
Another example of serializability testing.
(d) Precedence graph for schedule E.
(e) Precedence graph for schedule F.
(f) Precedence graph with two equivalent serial schedules.

If transactions are executed at will and then the resulting schedule is tested for seri-alizability, we must cancel the effect of the schedule if it turns out not to be serializ-able. This is a serious problem that makes this approach impractical. Hence, the approach taken in most practical systems is to determine methods or protocols that ensure serializability, without having to test the schedules themselves. The approach taken in most commercial DBMSs is to design **protocols** (sets of rules) that—if fol-lowed by *every* individual transaction or if enforced by a DBMS concurrency con-trol subsystem—will ensure serializability of *all schedules in which the transactions participate*.

Another problem appears here: When transactions are submitted continuously to the system, it is difficult to determine when a schedule begins and when it ends. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule *S*. Recall from Section 21.4.1 that the *committed projection C*(*S*) of a schedule *S* includes only the operations in *S* that belong to committed transactions. We can theoretically define a schedule *S* to be serializable if its committed projection *C*(*S*) is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

Next chapter we discuss a number of different concurrency control protocols that guarantee serializability. The most common technique, called *two-phase locking*, is based on locking data items to prevent concurrent transactions from interfering with one another, and enforcing an additional condition that guarantees serializ-ability. This is used in the majority of commercial DBMSs. Other protocols have been proposed;[14] these include *timestamp ordering*, where each transaction is assigned a unique timestamp and the protocol ensures that any conflicting opera-tions are executed in the order of the transaction timestamps; *multiversion protocols*, which are based on maintaining multiple versions of data items; and *optimistic* (also called *certification* or *validation*) *protocols*, which check for possible serializability violations after the transactions terminate but before they are permitted to commit.

1.5.4 View Equivalence and View Serializability

In Section 21.5.1 we defined the concepts of conflict equivalence of schedules and conflict serializability. Another less restrictive definition of equivalence of schedules is called *view equivalence*. This leads to

another definition of serializability called *view serializability*. Two schedules S and S are said to be **view equivalent** if the fol-lowing three conditions hold:

The same set of transactions participates in S and S , and S and S include the same operations of those transactions.

For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S .

If the operation $w_k(Y)$ of T_k is the last operation to write item Y in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S .

The idea behind view equivalence is that, as long as each read operation of a trans-action reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the data-base state should be the same at the end of both schedules. A schedule S is said to be **view serializable** if it is view equivalent to a serial schedule.

The definitions of conflict serializability and view serializability are similar if a con-dition known as the **constrained write assumption** (or **no blind writes**) holds on all transactions in the schedule. This condition states that any write operation $w_i(X)$ in T_i is preceded by a $r_i(X)$ in T_i and that the value written by $w_i(X)$ in T_i depends only on the value of X read by $r_i(X)$. This assumes that computation of the new value of X is a function $f(X)$ based on the old value of X read from the database. A **blind write** is a write operation in a transaction T on an item X that is not depen-dent on the value of X, so it is not preceded by a read of X in the transaction T.

The definition of view serializability is less restrictive than that of conflict serializ-ability under the **unconstrained write assumption**, where the value written by an operation $w_i(X)$ in T_i can be independent of its old value from the database. This is possible when *blind writes* are allowed, and it is illustrated by the following schedule S_g of three transactions T_1: $r_1(X)$; $w_1(X)$; T_2: $w_2(X)$; and T_3: $w_3(X)$:

S_g: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c_1; c_2; c_3;

In S_g the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T_2 and T_3 do not read the value of X. The schedule S_g is view serializable, since it is view equivalent to the serial schedule T_1, T_2, T_3. However, S_g is not conflict serializable, since it is not conflict equivalent to any serial schedule. It has been shown that any conflict-serializable schedule is also view serializable but not vice versa, as illustrated by the preceding example. There is an algorithm to test whether a schedule S is view serial-izable or not. However, the problem of testing for view serializability has been shown to be NP-hard, meaning that finding an efficient polynomial time algorithm for this problem is highly unlikely.

1.5.5 Other Types of Equivalence of Schedules

Serializability of schedules is sometimes considered to be too restrictive as a condi-tion for ensuring the correctness of concurrent executions. Some applications can produce schedules that are correct by

satisfying conditions less stringent than either conflict serializability or view serializability. An example is the type of transactions known as **debit-credit transactions**—for example, those that apply deposits and withdrawals to a data item whose value is the current balance of a bank account. The semantics of debit-credit operations is that they update the value of a data item X by either subtracting from or adding to the value of the data item. Because addi-tion and subtraction operations are commutative—that is, they can be applied in any order—it is possible to produce correct schedules that are not serializable. For example, consider the following transactions, each of which may be used to transfer an amount of money between two bank accounts:

T_1: $r_1(X)$; $X := X - 10$; $w_1(X)$; $r_1(Y)$; $Y := Y + 10$; $w_1(Y)$;
T_2: $r_2(Y)$; $Y := Y - 20$; $w_2(Y)$; $r_2(X)$; $X := X + 20$; $w_2(X)$;

Consider the following nonserializable schedule S_h for the two transactions:

S_h: $r_1(X)$; $w_1(X)$; $r_2(Y)$; $w_2(Y)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$;

With the additional knowledge, or **semantics**, that the operations between each $r_i(I)$ and $w_i(I)$ are commutative, we know that the order of executing the sequences con-sisting of (read, update, write) is not important as long as each (read, update, write) sequence by a particular transaction T_i on a particular item I is not interrupted by conflicting operations. Hence, the schedule S_h is considered to be correct even though it is not serializable. Researchers have been working on extending concur-rency control theory to deal with cases where serializability is considered to be too restrictive as a condition for correctness of schedules. Also, in certain domains of applications such as computer aided design (CAD) of complex systems like aircraft, design transactions last over a long time period. In such applications, more relaxed schemes of concurrency control have been proposed to maintain consistency of the database.

21.6 Transaction Support in SQL

In this section, we give a brief introduction to transaction support in SQL. There are many more details, and the newer standards have more commands for transaction processing. The basic definition of an SQL transaction is similar to our already defined concept of a transaction. That is, it is a logical unit of work and is guaran-teed to be atomic. A single SQL statement is always considered to be atomic—either it completes execution without an error or it fails and leaves the database unchanged.

With SQL, there is no explicit Begin_Transaction statement. Transaction initiation is done implicitly when particular SQL statements are encountered. However, every transaction must have an explicit end statement, which is either a COMMIT or a ROLLBACK. Every transaction has certain characteristics attributed to it. These characteristics are specified by a SET TRANSACTION statement in SQL. The charac-teristics are the *access mode*, the *diagnostic area size*, and the *isolation level*.

The **access mode** can be specified as READ ONLY or READ WRITE. The default is READ WRITE, unless the isolation level of READ UNCOMMITTED is specified (see below), in which case READ ONLY is assumed. A mode of READ WRITE allows select, update, insert, delete, and create commands to be executed. A mode of READ ONLY, as the name implies, is simply for data retrieval.

The **diagnostic area size** option, DIAGNOSTIC SIZE *n*, specifies an integer value *n*, which indicates the number of conditions that can be held simultaneously in the diagnostic area. These conditions supply feedback information (errors or excep-tions) to the user or program on the *n* most recently executed SQL statement.

The **isolation level** option is specified using the statement <isolation>, where the value for <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, or SERIALIZABLE.[15] The default isolation level is SERIALIZABLE, although some systems use READ COMMITTED as their default. The use of the term SERIALIZABLE here is based on not allowing violations that cause dirty read, unrepeatable read, and phantoms,[16] and it is thus not identical to the way serializability was defined earlier in Section 21.5. If a transaction executes at a lower isolation level than SERIALIZABLE, then one or more of the following three viola-tions may occur:

Dirty read. A transaction T_1 may read the update of a transaction T_2, which has not yet committed. If T_2 fails and is aborted, then T_1 would have read a value that does not exist and is incorrect.

Nonrepeatable read. A transaction T_1 may read a given value from a table. If another transaction T_2 later updates that value and T_1 reads that value again, T_1 will see a different value.

Phantoms. A transaction T_1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T_2 inserts a new row that also satisfies the WHERE-clause condition used in T_1, into the table used by T_1. If T_1 is repeated, then T_1 will see a phantom, a row that previously did not exist.

Table 1.1 summarizes the possible violations for the different isolation levels. An entry of *Yes* indicates that a violation is possible and an entry of *No* indicates that it is not possible. READ UNCOMMITTED is the most forgiving, and SERIALIZABLE is the most restrictive in that it avoids all three of the problems mentioned above.

A sample SQL transaction might look like the following:

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;

EXEC SQL SET TRANSACTION

READ WRITE

DIAGNOSTIC SIZE 5

ISOLATION LEVEL SERIALIZABLE;

EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)

VALUES ('Robert', 'Smith', '991004321', 2, 35000);

EXEC SQL UPDATE EMPLOYEE
```

```
SET Salary = Salary * 1.1 WHERE Dno = 2;

EXEC SQL COMMIT;

GOTO THE_END;

UNDO: EXEC SQL ROLLBACK;

THE_END: ... ;
```

The above transaction consists of first inserting a new row in the EMPLOYEE table and then updating the salary of all employees who work in department 2. If an error occurs on any of the SQL statements, the entire transaction is rolled back. This implies that any updated salary (by this transaction) would be restored to its previ-ous value and that the newly inserted row would be removed.

As we have seen, SQL provides a number of transaction-oriented features. The DBA or database programmers can take advantage of these options to try improving transaction performance by relaxing serializability if that is acceptable for their applications.

Table 1.1 Possible Violations Based on Isolation Levels as Defined in SQL

	ype of Violation		
lation Level	ty Read	nrepeatable Read	antom
AD UNCOMMITTED	;	;	;
AD COMMITTED		;	;
PEATABLE READ			;
RIALIZABLE			

21.7 Summary

In this chapter we discussed DBMS concepts for transaction processing. We intro-duced the concept of a database transaction and the operations relevant to transac-tion processing. We compared single-user systems to multiuser systems and then presented examples of how uncontrolled execution of concurrent transactions in a multiuser system can lead to incorrect results and database values. We also discussed the various types of failures that may occur during transaction execution.

Next we introduced the typical states that a transaction passes through during execu-tion, and discussed several concepts that are used in recovery and concurrency con-trol methods. The system log keeps track of database accesses, and the system uses this information to recover from failures. A transaction

either succeeds and reaches its commit point or it fails and has to be rolled back. A committed transaction has its changes permanently recorded in the database. We presented an overview of the desirable properties of transactions—atomicity, consistency preservation, isolation, and durability—which are often referred to as the ACID properties.

Then we defined a schedule (or history) as an execution sequence of the operations of several transactions with possible interleaving. We characterized schedules in terms of their recoverability. Recoverable schedules ensure that, once a transaction commits, it never needs to be undone. Cascadeless schedules add an additional con-dition to ensure that no aborted transaction requires the cascading abort of other transactions. Strict schedules provide an even stronger condition that allows a sim-ple recovery scheme consisting of restoring the old values of items that have been changed by an aborted transaction.

We defined equivalence of schedules and saw that a serializable schedule is equiva-lent to some serial schedule. We defined the concepts of conflict equivalence and view equivalence, which led to definitions for conflict serializability and view serial-izability. A serializable schedule is considered correct. We presented an algorithm for testing the (conflict) serializability of a schedule. We discussed why testing for serializability is impractical in a real system, although it can be used to define and verify concurrency control protocols, and we briefly mentioned less restrictive defi-nitions of schedule equivalence. Finally, we gave a brief overview of how transaction concepts are used in practice within SQL.

Review Questions

1.1. What is meant by the concurrent execution of database transactions in a multiuser system? Discuss why concurrency control is needed, and give informal examples.

1.2. Discuss the different types of failures. What is meant by catastrophic failure?

1.3. Discuss the actions taken by the read_item and write_item operations on a database.

1.4. Draw a state diagram and discuss the typical states that a transaction goes through during execution.

1.5. What is the system log used for? What are the typical kinds of records in a system log? What are transaction commit points, and why are they impor-tant?

1.6. Discuss the atomicity, durability, isolation, and consistency preservation properties of a database transaction.

1.7. What is a schedule (history)? Define the concepts of recoverable, cascadeless, and strict schedules, and compare them in terms of their recoverability.

1.8. Discuss the different measures of transaction equivalence. What is the differ-ence between conflict equivalence and view equivalence?

1.9. What is a serial schedule? What is a serializable schedule? Why is a serial schedule considered correct? Why is a serializable schedule considered cor-rect?

1.10. What is the difference between the constrained write and the unconstrained write assumptions? Which is more realistic?

1.11. Discuss how serializability is used to enforce concurrency control in a data-base system. Why is serializability sometimes considered too restrictive as a measure of correctness for schedules?

1.12. Describe the four levels of isolation in SQL.

1.13. Define the violations caused by each of the following: dirty read, nonrepeat-able read, and phantoms.

Exercises

1.14. Change transaction T_2 in Figure 21.2(b) to read

```
read_item(X);
    := X + M;
if X > 90 then exit
else write_item(X);
```

Discuss the final result of the different schedules in Figure 21.3(a) and (b), where $M = 2$ and $N = 2$, with respect to the following questions: Does adding the above condition change the final outcome? Does the outcome obey the implied consistency rule (that the capacity of X is 90)?

1.15. Repeat Exercise 21.14, adding a check in T_1 so that Y does not exceed 90.

1.16. Add the operation commit at the end of each of the transactions T_1 and T_2 in Figure 21.2, and then list all possible schedules for the modified transactions. Determine which of the schedules are recoverable, which are cascadeless, and which are strict.

1.17. List all possible schedules for transactions T_1 and T_2 in Figure 1.2, and determine which are conflict serializable (correct) and which are not.

1.18. How many *serial* schedules exist for the three transactions in Figure 1.8(a)? What are they? What is the total number of possible schedules?

1.19. Write a program to create all possible schedules for the three transactions in Figure 1.8(a), and to determine which of those schedules are conflict serial-izable and which are not. For each conflict-serializable schedule, your pro-gram should print the schedule and list all equivalent serial schedules.

1.20. Why is an explicit transaction end statement needed in SQL but not an explicit begin statement?

1.21. Describe situations where each of the different isolation levels would be use-ful for transaction processing.

1.22. Which of the following schedules is (conflict) serializable? For each serializ-able schedule, determine the equivalent serial schedules.
$r_1(X)$; $r_3(X)$; $w_1(X)$; $r_2(X)$; $w_3(X)$;
$r_1(X)$; $r_3(X)$; $w_3(X)$; $w_1(X)$; $r_2(X)$;

$r_3(X)$; $r_2(X)$; $w_3(X)$; $r_1(X)$; $w_1(X)$;

$r_3(X)$; $r_2(X)$; $r_1(X)$; $w_3(X)$; $w_1(X)$;

1.23. Consider the three transactions T_1, T_2, and T_3, and the schedules S_1 and S_2 given below. Draw the serializability (precedence) graphs for S_1 and S_2, and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).
T_1: r_1 (X); r_1 (Z); w_1 (X);
T_2: r_2 (Z); r_2 (Y); w_2 (Z); w_2 (Y);
T_3: r_3 (X); r_3 (Y); w_3 (Y);
S_1: r_1 (X); r_2 (Z); r_1 (Z); r_3 (X); r_3 (Y); w_1 (X); w_3 (Y); r_2 (Y); w_2 (Z); w_2 (Y);
S_2: r_1 (X); r_2 (Z); r_3 (X); r_1 (Z); r_2 (Y); r_3 (Y); w_1 (X); w_2 (Z); w_3 (Y); w_2 (Y);

1.24. Consider schedules S_3, S_4, and S_5 below. Determine whether each schedule is strict, cascadeless, recoverable, or nonrecoverable. (Determine the strictest recoverability condition that each schedule satisfies.)

S_3: r_1 (X); r_2 (Z); r_1 (Z); r_3 (X); r_3 (Y); w_1 (X); c_1; w_3 (Y); c_3; r_2 (Y); w_2 (Z); w_2 (Y); c_2;

S_4: r_1 (X); r_2 (Z); r_1 (Z); r_3 (X); r_3 (Y); w_1 (X); w_3 (Y); r_2 (Y); w_2 (Z); w_2 (Y); c_1; c_2; c_3;

S_5: r_1 (X); r_2 (Z); r_3 (X); r_1 (Z); r_2 (Y); r_3 (Y); w_1 (X); c_1; w_2 (Z); w_3 (Y); w_2 (Y); c_3; c_2;

2 Concurrency Control Techniques

In this chapter we discuss a number of concurrency control techniques that are used to ensure the nonin-terference or isolation property of concurrently executing transactions. Most of

these techniques ensure serializability of schedules—which we defined in Section 21.5—using **concurrency control protocols** (sets of rules) that guarantee serializ-ability. One important set of protocols—known as *two-phase locking protocols*— employ the technique of **locking** data items to prevent multiple transactions from accessing the items concurrently; a number of locking protocols are described in Sections 22.1 and 22.3.2. Locking protocols are used in most commercial DBMSs. Another set of concurrency control protocols use **timestamps**. A timestamp is a unique identifier for each transaction, generated by the system. Timestamps values are generated in the same order as the transaction start times. Concurrency control protocols that use timestamp ordering to ensure serializability are introduced in Section 22.2.

In Section 22.3 we discuss **multiversion** concurrency control proto-cols that use multiple versions of a data item. One multiversion protocol extends timestamp order to multiversion timestamp ordering (Section 22.3.1), and another extends two-phase locking (Section 22.3.2). In Section 22.4 we present a protocol based on the concept of **validation** or **certification** of a transaction after it executes its operations; these are sometimes called **optimistic protocols**, and also assume that multiple versions of a data item can exist.

Another factor that affects concurrency control is the **granularity** of the data items—that is, what portion of the database a data item represents. An item can be as small as a single attribute (field) value or as large as a disk block, or even a whole file or the entire database. We discuss granularity of items and a multiple granular-ity concurrency control protocol, which is an extension of two-phase locking, in Section 22.5. In Section 22.6 we describe concurrency control issues that arise when

indexes are used to process transactions, and in Section 22.7 we discuss some addi-tional concurrency control concepts. Section 22.8 summarizes the chapter.

It is sufficient to read Sections 22.1, 22.5, 22.6, and 22.7, and possibly 22.3.2, if your main interest is an introduction to the concurrency control techniques that are based on locking, which are used most often in practice. The other techniques are mainly of theoretical interest.

22.1 Two-Phase Locking Techniques for Concurrency Control

Some of the main techniques used to control concurrent execution of transactions are based on the concept of locking data items. A **lock** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it. Generally, there is one lock for each data item in the data-base. Locks are used as a means of synchronizing the access by concurrent transac-tions to the database items. In Section 22.1.1 we discuss the nature and types of locks. Then, in Section 22.1.2 we present protocols that use locking to guarantee serializability of transaction schedules. Finally, in Section 22.1.3 we describe two problems associated with the use of locks—deadlock and starvation—and show how these problems are handled in concurrency control protocols.

22.1.1 Types of Locks and System Lock Tables

Several types of locks are used in concurrency control. To introduce locking con-cepts gradually, first we discuss binary locks, which are simple, but are also *too restrictive for database concurrency control purposes*, and so are not used in practice. Then we discuss *shared/exclusive* locks—also known as *read/write* locks—which provide more general locking capabilities and are used in practical database

locking schemes. In Section 22.3.2 we describe an additional type of lock called a *certify lock*, and show how it can be used to improve performance of locking protocols.

Binary Locks. A **binary lock** can have two **states** or **values:** locked and unlocked (or 1 and 0, for simplicity). A distinct lock is associated with each database item *X*. If the value of the lock on *X* is 1, item *X cannot be accessed* by a database operation that requests the item. If the value of the lock on *X* is 0, the item can be accessed when requested, and the lock value is changed to 1. We refer to the current value (or state) of the lock associated with item *X* as **lock**(***X***).

Two operations, lock_item and unlock_item, are used with binary locking. A transaction requests access to an item *X* by first issuing a **lock_item(*X*)** operation. If LOCK(*X*) = 1, the transaction is forced to wait. If LOCK(*X*) = 0, it is set to 1 (the transaction **locks** the item) and the transaction is allowed to access item *X*. When the transaction is through using the item, it issues an **unlock_item(*X*)** operation, which sets LOCK(*X*) back to 0 (**unlocks** the item) so that *X* may be accessed by other transactions. Hence, a binary lock enforces **mutual exclusion** on the data item. A description of the lock_item(*X*) and unlock_item(*X*) operations is shown in Figure

```
lock_item(X):
B:  if LOCK(X) = 0          (* item is unlocked *)
        then LOCK(X) ← 1    (* lock the item *)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;            (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

Figure 22.1
Lock and unlock operations for binary locks.

Notice that the lock_item and unlock_item operations must be implemented as indi-visible units (known as **critical sections** in operating systems); that is, no interleav-ing should be allowed once a lock or unlock operation is started until the operation terminates or the transaction waits. In Figure 22.1, the wait command within the lock_item(*X*) operation is usually implemented by putting the transaction in a wait-ing queue for item *X* until *X* is unlocked and the transaction can be granted access to it. Other transactions that also want to access *X* are placed in the same queue. Hence, the wait command is considered to be outside the lock_item operation.

It is quite simple to implement a binary lock; all that is needed is a binary-valued variable, LOCK, associated with each data item *X* in the database. In its simplest form, each lock can be a record with three fields: <Data_item_name, LOCK, Locking_transaction> plus a queue for transactions that are waiting to access the item. The system needs to maintain *only these records for the items that are currently locked* in a **lock table**, which could be organized as a hash file on the item name. Items not in the lock table are considered to be unlocked. The DBMS has a **lock manager sub-system** to keep track of and control access to locks.

If the simple binary locking scheme described here is used, every transaction must obey the following rules:

A transaction *T* must issue the operation lock_item(*X*) before any read_item(*X*) or write_item(*X*) operations are performed in *T*.

A transaction *T* must issue the operation unlock_item(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.

A transaction *T* will not issue a lock_item(*X*) operation if it already holds the lock on item *X*.[1]

A transaction *T* will not issue an unlock_item(*X*) operation unless it already holds the lock on item *X*.

These rules can be enforced by the lock manager module of the DBMS. Between the lock_item(*X*) and unlock_item(*X*) operations in transaction *T*, *T* is said to **hold the lock** on item *X*. At most one transaction can hold the lock on a particular item. Thus no two transactions can access the same item concurrently.

Shared/Exclusive (or Read/Write) Locks. The preceding binary locking scheme is too restrictive for database items because at most, one transaction can hold a lock on a given item. We should allow several transactions to access the same item *X* if they all access *X* for *reading purposes only*. This is because read operations on the same item by different transactions are not conflicting (see Section 21.4.1). However, if a transaction is to write an item *X*, it must have exclusive access to *X*. For this purpose, a different type of lock called a **multiple-mode lock** is used. In this scheme—called **shared/exclusive** or **read/write** locks—there are three locking operations: read_lock(*X*), write_lock(*X*), and unlock(*X*). A lock associated with an item *X*, LOCK(*X*), now has three possible states: *read-locked*, *write-locked*, or *unlocked*. A **read-locked item** is also called **share-locked** because other transactions are allowed to read the item, whereas a **write-locked item** is called **exclusive-locked** because a single transaction exclusively holds the lock on the item.

One method for implementing the preceding operations on a read/write lock is to keep track of the number of transactions that hold a shared (read) lock on an item in the lock table. Each record in the lock table will have four fields: <Data_item_name, LOCK, No_of_reads, Locking_transaction(s)>. Again, to save space, the system needs to maintain lock records only for locked items in the lock table. The value (state) of LOCK is either read-locked or write-locked, suitably coded (if we assume no records are kept in the lock table for unlocked items). If LOCK(*X*)=write-locked, the value of locking_transaction(s) is a single transaction that holds the exclusive (write) lock on *X*. If LOCK(*X*)=read-locked, the value of locking transaction(s) is a list of one or more transactions that hold the shared (read) lock on *X*. The three operations read_lock(*X*), write_lock(*X*), and unlock(*X*) are described in Figure 22.2.[2] As before, each of the three locking operations should be considered indivisible; no interleav-ing should be allowed once one of the operations is started until either the opera-tion terminates by granting the lock or the transaction is placed in a waiting queue for the item.

When we use the shared/exclusive locking scheme, the system must enforce the fol-lowing rules:

A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.

A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed

```
read_lock(X):
B:  if LOCK(X) = "unlocked"
        then begin LOCK(X) ← "read-locked";
                no_of_reads(X) ← 1
                end
    else if LOCK(X) = "read-locked"
        then no_of_reads(X) ← no_of_reads(X) + 1
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
write_lock(X):
B:  if LOCK(X) = "unlocked"
        then LOCK(X) ← "write-locked"
    else begin
            wait (until LOCK(X) = "unlocked"
                and the lock manager wakes up the transaction);
            go to B
            end;
unlock (X):
    if LOCK(X) = "write-locked"
        then begin LOCK(X) ← "unlocked";
                wakeup one of the waiting transactions, if any
                end
    else if LOCK(X) = "read-locked"
        then begin
                no_of_reads(X) ← no_of_reads(X) −1;
                if no_of_reads(X) = 0
                    then begin LOCK(X) = "unlocked";
                            wakeup one of the waiting transactions, if any
                            end
                end;
```

Figure 22.2
Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.

A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.

A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X. This rule may be relaxed, as we discuss shortly.

A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X. This rule may also be relaxed, as we discuss shortly.

A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Conversion of Locks. Sometimes it is desirable to relax conditions 4 and 5 in the preceding list in order to allow **lock conversion**; that is, a transaction that already holds a lock on item X is allowed under certain conditions to **convert** the lock from one locked state to another. For example, it is possible for a transaction T to issue a read_lock(X) and then later to **upgrade** the lock by issuing a write_lock(X) operation. If T is the only transaction holding a read lock on X at the time it issues the write_lock(X)

operation, the lock can be upgraded; otherwise, the transaction must wait. It is also possible for a transaction T to issue a write_lock(X) and then later to **downgrade** the lock by issuing a read_lock(X) operation. When upgrading and downgrading of locks is used, the lock table must include transaction identifiers in the record structure for each lock (in the locking_transaction(s) field) to store the information on which transactions hold locks on the item. The descriptions of the read_lock(X) and write_lock(X) operations in Figure 22.2 must be changed appropri-ately to allow for lock upgrading and downgrading. We leave this as an exercise for the reader.

Using binary locks or read/write locks in transactions, as described earlier, does not guarantee serializability of schedules on its own. Figure 22.3 shows an example where the preceding locking rules are followed but a nonserializable schedule may result. This is because in Figure 22.3(a) the items Y in T_1 and X in T_2 were unlocked too early. This allows a schedule such as the one shown in Figure 22.3(c) to occur, which is not a serializable schedule and hence gives incorrect results. To guarantee serializability, we must follow *an additional protocol* concerning the positioning of locking and unlocking operations in every transaction. The best-known protocol, two-phase locking, is described in the next section.

2.1.2 Guaranteeing Serializability by Two-Phase Locking

A transaction is said to follow the **two-phase locking protocol** if *all* locking opera-tions (read_lock, write_lock) precede the *first* unlock operation in the transaction.[4] Such a transaction can be divided into two phases: an **expanding** or **growing (first) phase**, during which new locks on items can be acquired but none can be released; and a **shrinking (second) phase**, during which existing locks can be released but no new locks can be acquired. If lock conversion is allowed, then upgrading of locks (from read-locked to write-locked) must be done during the expanding phase, and downgrading of locks (from write-locked to read-locked) must be done in the

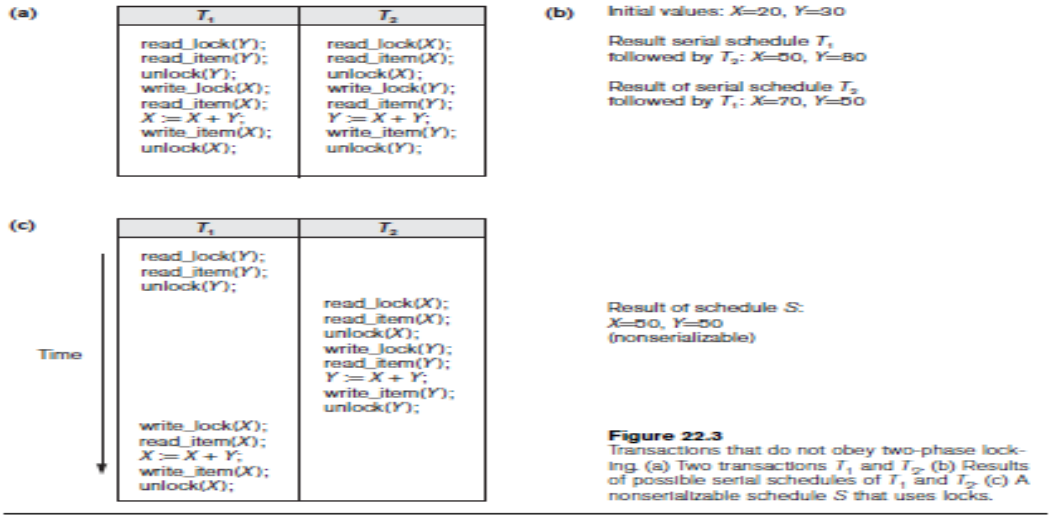


Figure 22.3
Transactions that do not obey two-phase lock-ing. (a) Two transactions T_1 and T_2. (b) Results of possible serial schedules of T_1 and T_2. (c) A nonserializable schedule S that uses locks.

shrinking phase. Hence, a read_lock(X) operation that downgrades an already held write lock on X can appear only in the shrinking phase.

Transactions T_1 and T_2 in Figure 22.3(a) do not follow the two-phase locking proto-col because the write_lock(X) operation follows the unlock(Y) operation in T_1, and similarly the write_lock(Y) operation follows the unlock(X) operation in T_2. If we

enforce two-phase locking, the transactions can be rewritten as T_1 and T_2 , as shown in Figure 22.4. Now, the schedule shown in Figure 22.3(c) is not permitted for T_1 and T_2 (with their modified order of locking and unlocking operations) under the rules of locking described in Section 22.1.1 because T_1 will issue its write_lock(X) *before* it unlocks item Y; consequently, when T_2 issues its read_lock(X), it is forced to wait until T_1 releases the lock by issuing an unlock (X) in the schedule.

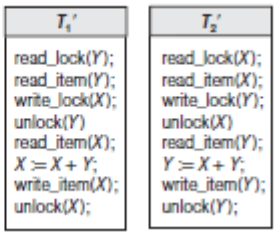


Figure 22.4
Transactions T_1' and T_2', which are the same as T_1 and T_2 in Figure 22.3, but follow the two-phase locking protocol. Note that they can produce a deadlock.

It can be proved that, if *every* transaction in a schedule follows the two-phase lock-ing protocol, the schedule is *guaranteed to be serializable*, obviating the need to test for serializability of schedules. The locking protocol, by enforcing two-phase lock-ing rules, also enforces serializability.

Two-phase locking may limit the amount of concurrency that can occur in a sched-ule because a transaction T may not be able to release an item X after it is through using it if T must lock an additional item Y later; or conversely, T must lock the additional item Y before it needs it so that it can release X. Hence, X must remain locked by T until all items that the transaction needs to read or write have been locked; only then can X be released by T. Meanwhile, another transaction seeking to access X may be forced to wait, even though T is done with X; conversely, if Y is locked earlier than it is needed, another transaction seeking to access Y is forced to wait even though T is not using Y yet. This is the price for guaranteeing serializabil-ity of all schedules without having to check the schedules themselves.

Although the two-phase locking protocol guarantees serializability (that is, every schedule that is permitted is serializable), it does not permit *all possible* serializable schedules (that is, some serializable schedules will be prohibited by the protocol).

Basic, Conservative, Strict, and Rigorous Two-Phase Locking. There are a number of variations of two-phase locking (2PL). The technique just described is known as **basic 2PL**. A variation known as **conservative 2PL** (or **static 2PL**) requires a transaction to lock all the items it accesses *before the transaction begins execution*, by **predeclaring** its *read-set* and *write-set*. Recall from Section 21.1.2 that the **read-set** of a transaction is the set of all items that the transaction reads, and the **write-set** is the set of all items that it writes. If any of the predeclared items needed cannot be locked, the transaction does not lock any item; instead, it waits until all the items are available for locking. Conservative 2PL is a deadlock-free protocol, as we will see in Section 22.1.3 when we discuss the deadlock problem. However, it is difficult to use in practice because of the need to predeclare the read-set and write-set, which is not possible in many situations.

In practice, the most popular variation of 2PL is **strict 2PL**, which guarantees strict schedules (see Section 21.4). In this variation, a transaction T does not release any of its exclusive (write) locks until *after* it commits or aborts. Hence, no other transac-tion can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. Strict 2PL is not deadlock-free. A more

restric-tive variation of strict 2PL is **rigorous 2PL**, which also guarantees strict schedules. In this variation, a transaction *T* does not release any of its locks (exclusive or shared) until after it commits or aborts, and so it is easier to implement than strict 2PL. Notice the difference between conservative and rigorous 2PL: the former must lock all its items *before it starts*, so once the transaction starts it is in its shrinking phase; the latter does not unlock any of its items until *after it terminates* (by com-mitting or aborting), so the transaction is in its expanding phase until it ends.

In many cases, the **concurrency control subsystem** itself is responsible for generat-ing the read_lock and write_lock requests. For example, suppose the system is to enforce the strict 2PL protocol. Then, whenever transaction *T* issues a read_item(*X*), the system calls the read_lock(*X*) operation on behalf of *T*. If the state of LOCK(*X*) is write_locked by some other transaction *T* , the system places *T* in the waiting queue for item *X*; otherwise, it grants the read_lock(*X*) request and permits the read_item(*X*) operation of *T* to execute. On the other hand, if transaction *T* issues a write_item(*X*), the system calls the write_lock(*X*) operation on behalf of *T*. If the state of LOCK(*X*) is write_locked or read_locked by some other transaction *T* , the system places *T* in the waiting queue for item *X*; if the state of LOCK(*X*) is read_locked and *T* itself is the only transaction holding the read lock on *X*, the system upgrades the lock to write_locked and permits the write_item(*X*) operation by *T*. Finally, if the state of LOCK(*X*) is unlocked, the system grants the write_lock(*X*) request and per-mits the write_item(*X*) operation to execute. After each action, the system must update its lock table appropriately.

2.1.3 Dealing with Deadlock and Starvation

Deadlock occurs when *each* transaction *T* in a set of *two or more transactions* is waiting for some item that is locked by some other transaction *T* in the set. Hence, each transaction in the set is in a waiting queue, waiting for one of the other trans-actions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock. A simple example is shown in Figure 22.5(a), where the two transactions T_1 and T_2 are deadlocked in a partial schedule; T_1 is in the waiting queue for *X*, which is locked by T_2 , while T_2 is in the waiting queue for *Y*, which is locked by T_1 . Meanwhile, neither T_1 nor T_2 nor any other transaction can access items *X* and *Y*.

Deadlock Prevention Protocols. One way to prevent deadlock is to use a **deadlock prevention protocol**.[5] One deadlock prevention protocol, which is used

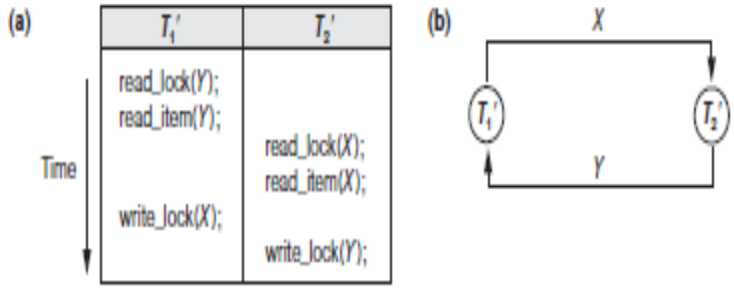


Figure 22.5
Illustrating the deadlock problem. (a) A partial schedule of T_1' and T_2' that is in a state of deadlock. (b) A wait-for graph for the partial schedule in (a).

in conservative two-phase locking, requires that every transaction lock *all the items it needs in advance* (which is generally not a practical assumption)—if any of the items cannot be obtained, none of the

items are locked. Rather, the transaction waits and then tries again to lock all the items it needs. Obviously this solution further limits concurrency. A second protocol, which also limits concurrency, involves *ordering all the items* in the database and making sure that a transaction that needs several items will lock them according to that order. This requires that the program-mer (or the system) is aware of the chosen order of the items, which is also not prac-tical in the database context.

A number of other deadlock prevention schemes have been proposed that make a decision about what to do with a transaction involved in a possible deadlock situa-tion: Should it be blocked and made to wait or should it be aborted, or should the transaction preempt and abort another transaction? Some of these techniques use the concept of **transaction timestamp** TS(T), which is a unique identifier assigned to each transaction. The timestamps are typically based on the order in which trans-actions are started; hence, if transaction T_1 starts before transaction T_2, then TS(T_1)

TS(T_2). Notice that the *older* transaction (which starts first) has the *smaller* time-stamp value. Two schemes that prevent deadlock are called *wait-die* and *wound-wait*. Suppose that transaction T_i tries to lock an item X but is not able to because X is locked by some other transaction T_j with a conflicting lock. The rules followed by these schemes are:

Wait-die. If TS(T_i) < TS(T_j), then (T_i older than T_j) T_i is allowed to wait; otherwise (T_i younger than T_j) abort T_i (T_i *dies*) and restart it later *with the same timestamp.*

Wound-wait. If TS(T_i) < TS(T_j), then (T_i older than T_j) abort T_j (T_i *wounds* T_j) and restart it later *with the same timestamp;* otherwise (T_i younger than T_j) T_i is allowed to wait.

In wait-die, an older transaction is allowed to *wait for a younger transaction*, whereas a younger transaction requesting an item held by an older transaction is aborted and restarted. The wound-wait approach does the opposite: A younger transaction is allowed to *wait for an older one*, whereas an older transaction requesting an item held by a younger transaction *preempts* the younger transaction by aborting it. Both schemes end up aborting the *younger* of the two transactions (the transaction that started later) that *may be involved* in a deadlock, assuming that this will waste less processing. It can be shown that these two techniques are *deadlock-free*, since in wait-die, transactions only wait for younger transactions so no cycle is created. Similarly, in wound-wait, transactions only wait for older transactions so no cycle is created. However, both techniques may cause some transactions to be aborted and restarted needlessly, even though those transactions may *never actually cause a deadlock*.

Another group of protocols that prevent deadlock do not require timestamps. These include the no waiting (NW) and cautious waiting (CW) algorithms. In the **no waiting algorithm**, if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not. In this case, no transaction ever waits, so no deadlock will occur. However, this scheme can cause transactions to abort and restart needlessly. The **cautious waiting** algorithm was proposed to try to reduce the number of needless aborts/restarts. Suppose that transaction T_i tries to lock an item X but is not able to do so because X is locked by some other transaction T_j with a conflicting lock. The cautious waiting rules are as follows:

Cautious waiting. If T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

It can be shown that cautious waiting is deadlock-free, because no transaction will ever wait for another blocked transaction. By considering the time $b(T)$ at which each blocked transaction T was blocked, if the two transactions T_i and T_j above both become blocked, and T_i is waiting for T_j, then $b(T_i) < b(T_j)$, since T_i can only wait for T_j at a time when T_j is not blocked itself. Hence, the blocking times form a total ordering on all blocked transactions, so no cycle that causes deadlock can occur.

Deadlock Detection. A second, more practical approach to dealing with deadlock is **deadlock detection**, where the system checks if a state of deadlock actually exists. This solution is attractive if we know there will be little interference among the transactions—that is, if different transactions will rarely access the same items at the same time. This can happen if the transactions are short and each transaction locks only a few items, or if the transaction load is light. On the other hand, if transac-tions are long and each transaction uses many items, or if the transaction load is quite heavy, it may be advantageous to use a deadlock prevention scheme.

A simple way to detect a state of deadlock is for the system to construct and main-tain a **wait-for graph**. One node is created in the wait-for graph for each transaction that is currently executing. Whenever a transaction T_i is waiting to lock an item X that is currently locked by a transaction T_j, a directed edge ($T_i \rightarrow T_j$) is created in the wait-for graph. When T_j releases the lock(s) on the items that T_i was waiting for, the directed edge is dropped from the wait-for graph. We have a state of dead-lock if and only if the wait-for graph has a cycle. One problem with this approach is the matter of determining *when* the system should check for a deadlock. One possi-bility is to check for a cycle every time an edge is added to the wait-for graph, but this may cause excessive overhead. Criteria such as the number of currently execut-ing transactions or the period of time several transactions have been waiting to lock items may be used instead to check for a cycle. Figure 22.5(b) shows the wait-for graph for the (partial) schedule shown in Figure 2.5(a).

If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted. Choosing which transactions to abort is known as **victim selec-tion**. The algorithm for victim selection should generally avoid selecting transac-tions that have been running for a long time and that have performed many updates, and it should try instead to select transactions that have not made many changes (younger transactions).

Timeouts. Another simple scheme to deal with deadlock is the use of **timeouts**. This method is practical because of its low overhead and simplicity. In this method, if a transaction waits for a period longer than a system-defined timeout period, the system assumes that the transaction may be deadlocked and aborts it—regardless of whether a deadlock actually exists or not.

Starvation. Another problem that may occur when we use locking is **starvation**, which occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally. This may occur if the waiting scheme for locked items is unfair, giving priority to some transactions over others. One solution for starvation is to have a fair waiting scheme, such as using a **first-come-first-served** queue; transactions are enabled to lock an item in the order in which they originally requested the lock. Another scheme allows some transac-tions to have priority over others but increases the priority of a transaction the longer it waits, until it eventually gets the highest priority and proceeds. Starvation can also occur because of victim selection if the algorithm selects the

same transac-tion as victim repeatedly, thus causing it to abort and never finish execution. The algorithm can use higher priorities for transactions that have been aborted multiple times to avoid this problem. The wait-die and wound-wait schemes discussed previ-ously avoid starvation, because they restart a transaction that has been aborted with its same original timestamp, so the possibility that the same transaction is aborted repeatedly is slim.

2.2 Concurrency Control Based on Timestamp Ordering

The use of locks, combined with the 2PL protocol, guarantees serializability of schedules. The serializable schedules produced by 2PL have their equivalent serial schedules based on the order in which executing transactions lock the items they acquire. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. Some transactions may be aborted and restarted because of the deadlock problem. A different approach that guarantees serializability involves using transaction timestamps to order transaction execution for an equiva-lent serial schedule.

2.2.1 Timestamps

Recall that a **timestamp** is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time*. We will refer to the timestamp of transaction *T* as **TS(*T*)**. Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.

Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction time-stamps are numbered 1, 2, 3, ... in this scheme. A computer counter has a finite max-imum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

2.2.2 The Timestamp Ordering Algorithm

The idea for this scheme is to order the transactions based on their timestamps. A schedule in which the transactions participate is then serializable, and the *only equivalent serial schedule permitted* has the transactions in order of their timestamp values. This is called **timestamp ordering (TO**). Notice how this differs from 2PL, where a schedule is serializable by being equivalent to some serial schedule allowed by the locking protocols. In timestamp ordering, however, the schedule is equivalent to the *particular serial order* corresponding to the order of the transaction time-stamps. The algorithm must ensure that, for each item accessed by *conflicting opera-tions* in the schedule, the order in which the item is accessed does not violate the timestamp order. To do this, the algorithm associates with each database item *X* two timestamp (**TS**) values:

read_TS(*X*). The **read timestamp** of item *X* is the largest timestamp among all the timestamps of transactions that have successfully read item *X*—that is, read_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has read *X* successfully.

write_TS(*X*). The **write timestamp** of item *X* is the largest of all the time-stamps of transactions that have successfully written item *X*—that is, write_TS(*X*) = TS(*T*), where *T* is the *youngest* transaction that has written *X* successfully.

Basic Timestamp Ordering (TO). Whenever some transaction *T* tries to issue a read_item(*X*) or a write_item(*X*) operation, the **basic TO** algorithm compares the timestamp of *T* with read_TS(*X*) and write_TS(*X*) to ensure that the timestamp order of transaction execution is not violated. If this order is violated, then transac-tion *T* is aborted and resubmitted to the system as a new transaction with a *new timestamp*. If *T* is aborted and rolled back, any transaction T_1 that may have used a value written by *T* must also be rolled back. Similarly, any transaction T_2 that may have used a value written by T_1 must also be rolled back, and so on. This effect is known as **cascading rollback** and is one of the problems associated with basic TO, since the schedules produced are not guaranteed to be recoverable. An *additional protocol* must be enforced to ensure that the schedules are recoverable, cascadeless, or strict. We first describe the basic TO algorithm here. The concurrency control algorithm must check whether conflicting operations violate the timestamp order-ing in the following two cases:

Whenever a transaction *T* issues a write_item(*X*) operation, the following is checked:

If read_TS(*X*) > TS(*T*) or if write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the operation. This should be done because some *younger* transaction with a timestamp greater than TS(*T*)—and hence *after T* in the timestamp ordering—has already read or written the value of item *X* before *T* had a chance to write *X*, thus violating the timestamp ordering.

If the condition in part (a) does not occur, then execute the write_item(*X*) operation of *T* and set write_TS(*X*) to TS(*T*).

Whenever a transaction *T* issues a read_item(*X*) operation, the following is checked:

If write_TS(*X*) > TS(*T*), then abort and roll back *T* and reject the opera-tion. This should be done because some younger transaction with time-stamp greater than TS(*T*)—and hence *after T* in the timestamp

ordering—has already written the value of item *X* before *T* had a chance to read *X*.

If write_TS(*X*) ? TS(*T*), then execute the read_item(*X*) operation of *T* and set read_TS(*X*) to the *larger* of TS(*T*) and the current read_TS(*X*).

Whenever the basic TO algorithm detects two *conflicting operations* that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. The schedules produced by basic TO are hence guaranteed to be *conflict serializable*, like the 2PL protocol. However, some schedules are possible under each protocol that are not allowed under the other. Thus, *neither* protocol allows *all possible* serializable schedules. As mentioned earlier, deadlock does not occur with timestamp ordering. However, cyclic restart (and hence starvation) may occur if a transaction is continually aborted and restarted.

Strict Timestamp Ordering (TO). A variation of basic TO called **strict TO** ensures that the schedules are both **strict** (for easy recoverability) and (conflict) serializable. In this variation, a transaction T that issues a read_item(X) or write_item(X) such that TS(T) > write_TS(X) has its read or write operation *delayed* until the transaction T that *wrote* the value of X (hence TS(T) = write_TS(X)) has committed or aborted. To implement this algorithm, it is necessary to simulate thelocking of an item X that has been written by transaction T until T is either com-mitted or aborted. This algorithm *does not cause deadlock*, since T waits for T only if TS(T) > TS(T).

Thomas's Write Rule. A modification of the basic TO algorithm, known as **Thomas's write rule**, does not enforce conflict serializability, but it rejects fewer write operations by modifying the checks for the write_item(X) operation as follows:

If read_TS(X) > TS(T), then abort and roll back T and reject the operation.

If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T)—and hence after T in the timestamp ordering—has already written the value of X. Thus, we must ignore the write_item(X) operation of T because it is already outdated and obsolete. Notice that any conflict arising from this situation would be detected by case (1).

If neither the condition in part (1) nor the condition in part (2) occurs, then execute the write_item(X) operation of T and set write_TS(X) to TS(T).

2.3 Multiversion Concurrency Control Techniques

Other protocols for concurrency control keep the old values of a data item when the item is updated. These are known as **multiversion concurrency control**, because several versions (values) of an item are maintained. When a transaction requires access to an item, an *appropriate* version is chosen to maintain the serializability of the currently executing schedule, if possible. The idea is that some read operations that would be rejected in other techniques can still be accepted by reading an *older version* of the item to maintain serializability. When a transaction writes an item, it writes a *new version* and the old version(s) of the item are retained. Some multiver-sion concurrency control algorithms use the concept of view serializability rather than conflict serializability.

An obvious drawback of multiversion techniques is that more storage is needed to maintain multiple versions of the database items. However, older versions may have to be maintained anyway—for example, for recovery purposes. In addition, some database applications require older versions to be kept to maintain a history of the evolution of data item values. The extreme case is a *temporal database* which keeps track of all changes and the times at which they occurred. In such cases, there is no additional storage penalty for multiversion techniques, since older versions are already maintained.

2.3.1 Multiversion Technique Based on Timestamp Ordering

In this method, several versions X_1, X_2, ..., X_k of each data item X are maintained. For *each version*, the value of version X_i and the following two timestamps are kept:

read_TS(X_i). The **read timestamp** of X_i is the largest of all the timestamps of transactions that have successfully read version X_i.
write_TS(X_i). The **write timestamp** of X_i is the timestamp of the transac-tion that wrote the value of version X_i.

Whenever a transaction T is allowed to execute a write_item(X) operation, a new ver-sion X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T). Correspondingly, when a transaction T is allowed to read the value of version X_i, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following rules are used:

If transaction T issues a write_item(X) operation, and version i of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).

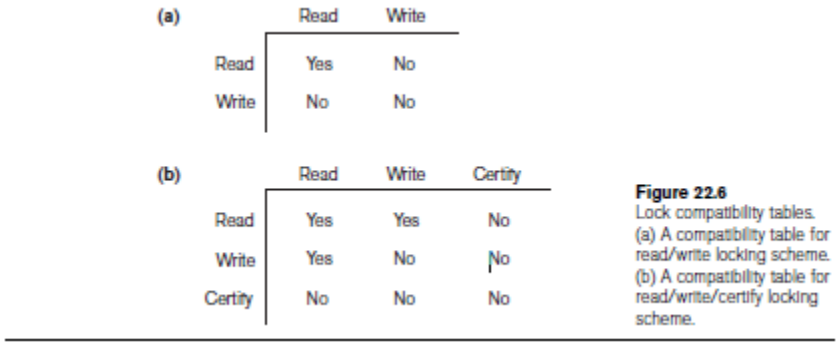
If transaction T issues a read_item(X) operation, find the version i of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As we can see in case 2, a read_item(X) is always successful, since it finds the appro-priate version X_i to read based on the write_TS of the various existing versions of X. In case 1, however, transaction T may be aborted and rolled back. This happens if T attempts to write a version of X that should have been read by another transaction T whose timestamp is read_TS(X_i); however, T has already read version X_i, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of X, written by transaction T, is created. Notice that if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

22.3.2 Multiversion Two-Phase Locking Using Certify Locks

In this multiple-mode locking scheme, there are *three locking modes* for an item: read, write, and *certify*, instead of just the two modes (read, write) discussed previ-ously. Hence, the state of LOCK(X) for an item X can be one of read-locked, write-locked, certify-locked, or unlocked. In the standard locking scheme, with only read and write locks (see Section 22.1.1), a write lock is an exclusive lock. We can describe the relationship between read and write locks in the standard scheme by means of the **lock compatibility table** shown in Figure 22.6(a). An entry of *Yes*

means that if a transaction T holds the type of lock specified in the column header



(a)

	Read	Write
Read	Yes	No
Write	No	No

(b)

	Read	Write	Certify
Read	Yes	Yes	No
Write	Yes	No	No
Certify	No	No	No

Figure 22.6
Lock compatibility tables. (a) A compatibility table for read/write locking scheme. (b) A compatibility table for read/write/certify locking scheme.

on item X and if transaction T requests the type of lock specified in the row header on the same item X, then T *can obtain the lock* because the locking modes are com-patible. On the other hand, an entry of *No* in the table indicates that the locks are not compatible, so T *must wait* until T *releases* the lock.

In the standard locking scheme, once a transaction obtains a write lock on an item, no other transactions can access that item. The idea behind multiversion 2PL is to allow other transactions T to read an item X while a single transaction T holds a write lock on X. This is accomplished by allowing *two versions* for each item X; one version must always have been written by some committed transaction. The second version X is created when a transaction T acquires a write lock on the item. Other transactions can continue to read the *committed version* of X while T holds the write lock. Transaction T can write the value of X as needed, without affecting the value of the committed version X. However, once T is ready to commit, it must obtain a **certify lock** on all items that it currently holds write locks on before it can commit. The certify lock is not compatible with read locks, so the transaction may have to delay its commit until all its write-locked items are released by any reading transac-tions in order to obtain the certify locks. Once the certify locks—which are exclusive locks—are acquired, the committed version X of the data item is set to the value of version X , version X is discarded, and the certify locks are then released. The lock compatibility table for this scheme is shown in Figure 2.6(b).

In this multiversion 2PL scheme, reads can proceed concurrently with a single write operation—an arrangement not permitted under the standard 2PL schemes. The cost is that a transaction may have to delay its commit until it obtains exclusive cer-tify locks on *all the items* it has updated. It can be shown that this scheme avoids cas-cading aborts, since transactions are only allowed to read the version X that was written by a committed transaction. However, deadlocks may occur if upgrading of a read lock to a

2.4 Validation (Optimistic) Concurrency Control Techniques

In all the concurrency control techniques we have discussed so far, a certain degree of checking is done *before* a database operation can be executed. For example, in locking, a check is done to determine whether the item being accessed is locked. In timestamp ordering, the transaction timestamp is checked against the read and write timestamps of the item. Such checking represents overhead during transac-tion execution, with the effect of slowing down the transactions.

In **optimistic concurrency control techniques**, also known as **validation** or **certification techniques**, *no checking* is done while the transaction is executing. Several theoretical concurrency control methods are based on the validation tech-nique. We will describe only one scheme here. In this scheme, updates in the trans-action are *not* applied directly to the database items until the transaction reaches its end. During transaction execution, all updates are applied to *local copies* of the data items that are kept for the transaction.[6] At the end of transaction execution, a **validation phase** checks whether any of the transaction's updates violate serializ-ability. Certain information needed by the validation phase must be kept by the sys-tem. If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.

There are three phases for this concurrency control protocol:

Read phase. A transaction can read values of committed data items from the database. However, updates are applied only to local copies (versions) of the data items kept in the transaction workspace.

Validation phase. Checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.

Write phase. If the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the trans-action is restarted.

The idea behind optimistic concurrency control is to do all the checks at once; hence, transaction execution proceeds with a minimum of overhead until the vali-dation phase is reached. If there is little interference among transactions, most will be validated successfully. However, if there is much interference, many transactions that execute to completion will have their results discarded and must be restarted later. Under these circumstances, optimistic techniques do not work well. The tech-niques are called *optimistic* because they assume that little interference will occur and hence that there is no need to do checking during transaction execution.

The optimistic protocol we describe uses transaction timestamps and also requires that the write_sets and read_sets of the transactions be kept by the system. Additionally, *start* and *end* times for some of the three phases need to be kept for each transaction. Recall that the write_set of a transaction is the set of items it writes, and the read_set is the set of items it reads. In the validation phase for transaction T_i, the protocol checks that T_i does not interfere with any committed transactions or with any other transactions currently in their validation phase. The validation phase for T_i checks that, for *each* such transaction T_j that is either committed or is in its validation phase, *one* of the following conditions holds:

Transaction T_j completes its write phase before T_i starts its read phase.
T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.

Both the read_set and write_set of T_i have no items in common with the write_set of T_j, and T_j completes its read phase before T_i completes its read phase.

When validating transaction T_i, the first condition is checked first for each transac-tion T_j, since (1) is the simplest condition to check. Only if condition 1 is false is condition 2 checked, and only if (2) is false is condition 3—the most complex to evaluate—checked. If any one of these three conditions holds, there is no interfer-ence and T_i is validated successfully. If *none* of these three conditions holds, the val-idation of transaction T_i fails and it is aborted and restarted later because interference *may* have occurred.

2.5 Granularity of Data Items and Multiple Granularity Locking

All concurrency control techniques assume that the database is formed of a number of named data items. A database item could be chosen to be one of the following:

A database record

A field value of a database record

A disk block

A whole file

The whole database

The granularity can affect the performance of concurrency control and recovery. In Section 22.5.1, we discuss some of the tradeoffs with regard to choosing the granu-larity level used for locking, and in Section 22.5.2 we discuss a multiple granularity locking scheme, where the granularity level (size of the data item) may be changed dynamically.

22.5.1 Granularity Level Considerations for Locking

The size of data items is often called the **data item granularity**. *Fine granularity* refers to small item sizes, whereas *coarse granularity* refers to large item sizes. Several tradeoffs must be considered in choosing the data item size. We will discuss data item size in the context of locking, although similar arguments can be made for other concurrency control techniques. First, notice that the larger the data item size is, the lower the degree of concurrency permitted. For example, if the data item size is a disk block, a transaction *T* that needs to lock a record *B* must lock the whole disk block *X* that contains *B* because a lock is associated with the whole data item (block). Now, if another transaction *S* wants to lock a different record *C* that happens to reside in the same block *X* in a conflicting lock mode, it is forced to wait. If the data item size was a single record, transaction *S* would be able to proceed, because it would be locking a different data item (record).

On the other hand, the smaller the data item size is, the more the number of items in the database. Because every item is associated with a lock, the system will have a larger number of active locks to be handled by the lock manager. More lock and unlock operations will be performed, causing a higher overhead. In addition, more storage space will be required for the lock table. For timestamps, storage is required for

the read_TS and write_TS for each data item, and there will be similar overhead for handling a large number of items.

Given the above tradeoffs, an obvious question can be asked: What is the best item size? The answer is that it *depends on the types of transactions involved*. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity be one record. On the other hand, if a transaction typically accesses many records in the same file, it may be better to have block or file granularity so that the transaction will consider all those records as one (or a few) data items.

2.5.2 Multiple Granularity Level Locking

Since the best granularity size depends on the given transaction, it seems appropri-ate that a database system should support multiple levels of granularity, where the granularity level can be different for various mixes of transactions. Figure 22.7 shows a simple granularity hierarchy with a database containing two files, each file containing several disk pages, and each page containing several records. This can be used to illustrate a **multiple granularity level** 2PL protocol, where a lock can be requested at any level. However, additional types of locks will be needed to support such a protocol efficiently.

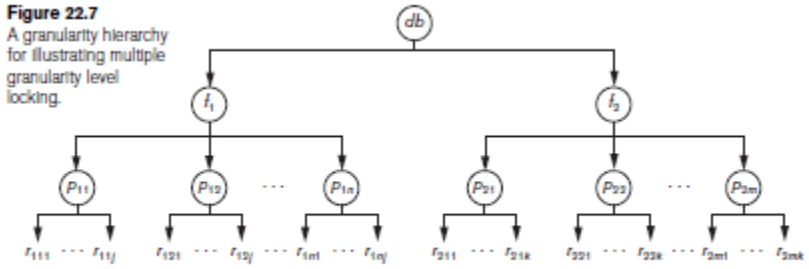


Figure 22.7
A granularity hierarchy for illustrating multiple granularity level locking.

Consider the following scenario, with only shared and exclusive lock types, that refers to the example in Figure 22.7. Suppose transaction T_1 wants to update *all the records* in file f_1, and T_1 requests and is granted an exclusive lock for f_1. Then all of f_1's pages (p_{11} through p_{1n})—and the records contained on those pages—are locked in exclu-sive mode. This is beneficial for T_1 because setting a single file-level lock is more effi-cient than setting n page-level locks or having to lock each individual record. Now suppose another transaction T_2 only wants to read record r_{1nj} from page p_{1n} of file f_1; then T_2 would request a shared record-level lock on r_{1nj}. However, the database sys-tem (that is, the transaction manager or more specifically the lock manager) must verify the compatibility of the requested lock with already held locks. One way to ver-ify this is to traverse the tree from the leaf r_{1nj} to p_{1n} to f_1 to *db*. If at any time a con-flicting lock is held on any of those items, then the lock request for r_{1nj} is denied and T_2 is blocked and must wait. This traversal would be fairly efficient.

However, what if transaction T_2's request came *before* transaction T_1's request? In this case, the shared record lock is granted to T_2 for r_{1nj}, but when T_1's file-level lock is requested, it is quite difficult for the lock manager to check all nodes (pages and records) that are descendants of node f_1 for a lock conflict. This would be very inef-ficient and would defeat the purpose of having multiple granularity level locks.

To make multiple granularity level locking practical, additional types of locks, called **intention locks**, are needed. The idea behind intention locks is for a transaction to indicate, along the path from the root to the desired node, what type of lock (shared or exclusive) it will require from one of the node's descendants. There are three types of intention locks:

Intention-shared (IS) indicates that one or more shared locks will be requested on some descendant node(s).

Intention-exclusive (IX) indicates that one or more exclusive locks will be requested on some descendant node(s).

Shared-intention-exclusive (SIX) indicates that the current node is locked in shared mode but that one or more exclusive locks will be requested on some descendant node(s).

The compatibility table of the three intention locks, and the shared and exclusive locks, is shown in Figure 22.8. Besides the introduction of the three types of inten-tion locks, an appropriate locking protocol must be used. The **multiple granularity locking (MGL)** protocol consists of the following rules:

The lock compatibility (based on Figure 22.8) must be adhered to.

The root of the tree must be locked first, in any mode.

A node *N* can be locked by a transaction *T* in S or IS mode only if the parent node *N* is already locked by transaction *T* in either IS or IX mode.

A node *N* can be locked by a transaction *T* in X, IX, or SIX mode only if the parent of node *N* is already locked by transaction *T* in either IX or SIX mode.

A transaction *T* can lock a node only if it has not unlocked any node (to enforce the 2PL protocol).

	IS	IX	S	SIX	X
IS	Yes	Yes	Yes	Yes	No
IX	Yes	Yes	No	No	No
S	Yes	No	Yes	No	No
SIX	Yes	No	No	No	No
X	No	No	No	No	No

Figure 22.8
Lock compatibility matrix for multiple granularity locking.

A transaction *T* can unlock a node, *N*, only if none of the children of node *N* are currently locked by *T*.

Rule 1 simply states that conflicting locks cannot be granted. Rules 2, 3, and 4 state the conditions when a transaction may lock a given node in any of the lock modes. Rules 5 and 6 of the MGL protocol enforce 2PL rules to produce serializable sched-ules. To illustrate the MGL protocol with the database hierarchy in Figure 22.7, con-sider the following three transactions:

T_1 wants to update record r_{111} and record r_{211}.
T_2 wants to update all records on page p_{12}.

T_3 wants to read record r_{11j} and the entire f_2 file.

Figure 22.9 shows a possible serializable schedule for these three transactions. Only the lock and unlock operations are shown. The notation <lock_type>(<item>) is used to display the locking operations in the schedule.

The multiple granularity level protocol is especially suited when processing a mix of transactions that include (1) short transactions that access only a few items (records or fields) and (2) long transactions that access entire files. In this environment, less transaction blocking and less locking overhead is incurred by such a protocol when compared to a single level granularity locking approach.

2.6 Using Locks for Concurrency Control in Indexes

Two-phase locking can also be applied to indexes (see Chapter 18), where the nodes of an index correspond to disk pages. However, holding locks on index pages until the shrinking phase of 2PL could cause an undue amount of transaction blocking because searching an index always *starts at the root*. Therefore, if a transaction wants to insert a record (write operation), the root would be locked in exclusive mode, so all other conflicting lock requests for the index must wait until the transaction enters its shrinking phase. This blocks all other transactions from accessing the index, so in practice other approaches to locking an index must be used.

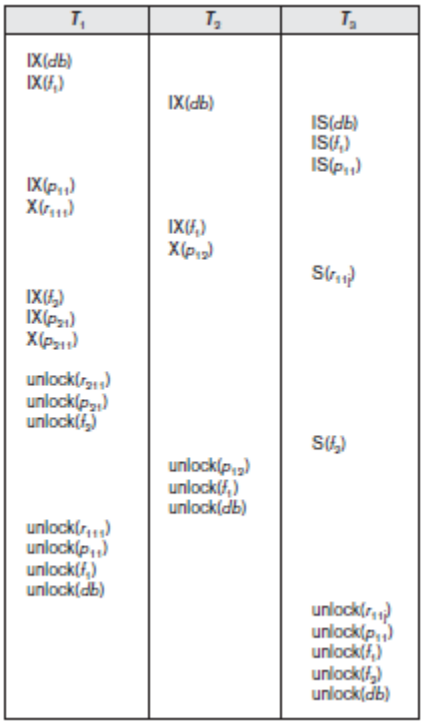


Figure 22.9
Lock operations to illustrate a serializable schedule.

The tree structure of the index can be taken advantage of when developing a con-currency control scheme. For example, when an index search (read operation) is being executed, a path in the tree is traversed from the root to a leaf. Once a lower-level node in the path has been accessed, the higher-level nodes in that path will not be used again. So once a read lock on a child node is obtained, the lock on the par-ent can be released. When an insertion is being applied to a leaf node (that is, when a key and a pointer are inserted), then a specific leaf node must be locked in exclu-sive mode. However, if that node is not full, the insertion will not cause changes to higher-level index nodes, which implies that they need not be locked exclusively.

A conservative approach for insertions would be to lock the root node in exclusive mode and then to access the appropriate child node of the root. If the child node is not full, then the lock on the root node can be released. This approach can be applied all the way down the tree to the leaf, which is typically three or four levels from the root. Although exclusive locks are held, they are soon released. An alterna-tive, more **optimistic approach** would be to request and hold *shared* locks on the nodes leading to the leaf node, with an *exclusive* lock on the leaf. If the insertion causes the leaf to split, insertion will propagate to one or more higher-level nodes. Then, the locks on the higher-level nodes can be upgraded to exclusive mode.

Another approach to index locking is to use a variant of the B^+-tree, called the **B-link tree**. In a B-link tree, sibling nodes on the same level are linked at every level. This allows shared locks to be used when requesting a page and requires that the lock be released before accessing the child node. For an insert operation, the shared lock on a node would be upgraded to exclusive mode. If a split occurs, the parent node must be relocked in exclusive mode. One complication is for search operations executed concurrently with the update. Suppose that a concurrent update operation follows the same path as the search, and inserts a new entry into the leaf node. Additionally, suppose that the insert causes that leaf node to split. When the insert is done, the search process resumes, following the pointer to the desired leaf, only to find that the key it is looking for is not present because the split has moved that key into a new leaf node, which would be the *right sibling* of the original leaf node. However, the search process can still succeed if it follows the pointer (link) in the original leaf node to its right sibling, where the desired key has been moved.

Handling the deletion case, where two or more nodes from the index tree merge, is also part of the B-link tree concurrency protocol. In this case, locks on the nodes to be merged are held as well as a lock on the parent of the two nodes to be merged.

2.7 Other Concurrency Control Issues

In this section we discuss some other issues relevant to concurrency control. In Section 22.7.1, we discuss problems associated with insertion and deletion of records and the so-called *phantom problem*, which may occur when records are inserted. This problem was described as a potential problem requiring a concur-rency control measure in Section 21.6. In Section 22.7.2 we discuss problems that may occur when a transaction outputs some data to a monitor before it commits, and then the transaction is later aborted.

2.7.1 Insertion, Deletion, and Phantom Records

When a new data item is **inserted** in the database, it obviously cannot be accessed until after the item is created and the insert operation is completed. In a locking environment, a lock for the item can be created and set to exclusive (write) mode; the lock can be released at the same time as other write locks would be released, based on the concurrency control protocol being used. For a timestamp-based protocol, the read and write timestamps of the new item are set to the timestamp of the creating transaction.Next, consider a **deletion operation** that is applied on an existing data item. For locking protocols, again an exclusive (write) lock must be obtained before the trans-action can delete the item. For timestamp ordering, the protocol must ensure that no later transaction has read or written the item before allowing the item to be deleted.

A situation known as the **phantom problem** can occur when a new record that is being inserted by some transaction *T* satisfies a condition that a set of records accessed by another transaction *T* must satisfy. For example, suppose that transac-tion *T* is inserting a new EMPLOYEE record whose Dno = 5, while transaction *T* is accessing all EMPLOYEE records whose Dno = 5 (say, to add up all their Salary values to calculate the personnel budget for department 5). If the equivalent serial order is *T* followed by *T* , then *T* must read the new EMPLOYEE record and include its Salary in the sum calculation. For the equivalent serial order *T* followed by *T*, the new salary should not be included. Notice that although the transactions logically con-flict, in the latter case there is really no record (data item) in common between the two transactions, since *T* may have locked all the records with Dno = 5 *before T* inserted the new record. This is because the record that causes the conflict is a **phantom record** that has suddenly appeared in the database on being inserted. If other operations in the two transactions conflict, the conflict due to the phantom record may not be recognized by the concurrency control protocol.

One solution to the phantom record problem is to use **index locking**, as discussed in Section 22.6. Recall from Chapter 18 that an index includes entries that have an attribute value, plus a set of pointers to all records in the file with that value. For example, an index on Dno of EMPLOYEE would include an entry for each distinct Dno value, plus a set of pointers to all EMPLOYEE records with that value. If the index entry is locked before the record itself can be accessed, then the conflict on the phantom record can be detected, because transaction *T* would request a read lock on the *index entry* for Dno = 5, and *T* would request a write lock on the same entry *before* they could place the locks on the actual records. Since the index locks conflict, the phantom conflict would be detected.

A more general technique, called **predicate locking**, would lock access to all records that satisfy an arbitrary *predicate* (condition) in a similar manner; however, predi-cate locks have proved to be difficult to implement efficiently.

2.7.2 Interactive Transactions

Another problem occurs when interactive transactions read input and write output to an interactive device, such as a monitor screen, before they are committed. The problem is that a user can input a value of a data item to a transaction *T* that is based on some value written to the screen by transaction *T* , which may not have committed. This dependency between *T* and *T* cannot be modeled by the system concurrency control method, since it is only based on the user interacting with the two transactions.

An approach to dealing with this problem is to postpone output of transactions to the screen until they have committed.

2.7.3 Latches

Locks held for a short duration are typically called **latches**. Latches do not follow the usual concurrency control protocol such as two-phase locking. For example, a latch can be used to guarantee the physical integrity of a page when that page is being written from the buffer to disk. A latch would be acquired for the page, the page written to disk, and then the latch released.

22.8 Summary

In this chapter we discussed DBMS techniques for concurrency control. We started by discussing lock-based protocols, which are by far the most commonly used in practice. We described the two-phase locking (2PL) protocol and a number of its variations: basic 2PL, strict 2PL, conservative 2PL, and rigorous 2PL. The strict and rigorous variations are more common because of their better recoverability proper-ties. We introduced the concepts of shared (read) and exclusive (write) locks, and showed how locking can guarantee serializability when used in conjunction with the two-phase locking rule. We also presented various techniques for dealing with the deadlock problem, which can occur with locking. In practice, it is common to use timeouts and deadlock detection (wait-for graphs).

We presented other concurrency control protocols that are not used often in prac-tice but are important for the theoretical alternatives they show for solving this problem. These include the timestamp ordering protocol, which ensures serializ-ability based on the order of transaction timestamps. Timestamps are unique, system-generated transaction identifiers. We discussed Thomas's write rule, which improves performance but does not guarantee conflict serializability. The strict timestamp ordering protocol was also presented. We discussed two multiversion protocols, which assume that older versions of data items can be kept in the data-base. One technique, called multiversion two-phase locking (which has been used in practice), assumes that two versions can exist for an item and attempts to increase concurrency by making write and read locks compatible (at the cost of introducing an additional certify lock mode). We also presented a multiversion protocol based on timestamp ordering, and an example of an optimistic protocol, which is also known as a certification or validation protocol.

Then we turned our attention to the important practical issue of data item granu-larity. We described a multigranularity locking protocol that allows the change of granularity (item size) based on the current transaction mix, with the goal of improving the performance of concurrency control. An important practical issue was then presented, which is to develop locking protocols for indexes so that indexes do not become a hindrance to concurrent access. Finally, we introduced the phan-tom problem and problems with interactive transactions, and briefly described the concept of latches and how it differs from locks.

Review Questions

2.1. What is the two-phase locking protocol? How does it guarantee serializabil-ity?

2.2. What are some variations of the two-phase locking protocol? Why is strict or rigorous two-phase locking often preferred?

2.3. Discuss the problems of deadlock and starvation, and the different approaches to dealing with these problems.

2.4. Compare binary locks to exclusive/shared locks. Why is the latter type of locks preferable?

2.5. Describe the wait-die and wound-wait protocols for deadlock prevention.

2.6. Describe the cautious waiting, no waiting, and timeout protocols for dead-lock prevention.

2.7. What is a timestamp? How does the system generate timestamps?

2.8. Discuss the timestamp ordering protocol for concurrency control. How does strict timestamp ordering differ from basic timestamp ordering?

2.9. Discuss two multiversion techniques for concurrency control.

2.10. What is a certify lock? What are the advantages and disadvantages of using certify locks?

2.11. How do optimistic concurrency control techniques differ from other con-currency control techniques? Why are they also called validation or certifica-tion techniques? Discuss the typical phases of an optimistic concurrency control method.

2.12. How does the granularity of data items affect the performance of concur-rency control? What factors affect selection of granularity size for data items?

2.13. What type of lock is needed for insert and delete operations?

2.14. What is multiple granularity locking? Under what circumstances is it used?

2.15. What are intention locks?

2.16. When are latches used?

2.17. What is a phantom record? Discuss the problem that a phantom record can cause for concurrency control.

2.18. How does index locking resolve the phantom problem?

2.19. What is a predicate lock?

Exercises

2.20. Prove that the basic two-phase locking protocol guarantees conflict serializ-ability of schedules. (*Hint*: Show that if a serializability graph for a schedule has a cycle, then at least one of the transactions participating in the schedule does not obey the two-phase locking protocol.)

2.21. Modify the data structures for multiple-mode locks and the algorithms for read_lock(*X*), write_lock(*X*), and unlock(*X*) so that upgrading and downgrad-ing of locks are possible. (*Hint*: The lock needs to check the transaction id(s) that hold the lock, if any.)

2.22. Prove that strict two-phase locking guarantees strict schedules.

2.23. Prove that the wait-die and wound-wait protocols avoid deadlock and star-vation.

2.24. Prove that cautious waiting avoids deadlock.

2.25. Apply the timestamp ordering algorithm to the schedules in Figure 21.8(b) and (c), and determine whether the algorithm will allow the execution of the schedules.

2.26. Repeat Exercise 22.25, but use the multiversion timestamp ordering method.

2.27. Why is two-phase locking not used as a concurrency control method for indexes such as B^+-trees?

2.28. The compatibility matrix in Figure 22.8 shows that IS and IX locks are com-patible. Explain why this is valid.

2.29. The MGL protocol states that a transaction *T* can unlock a node *N*, only if none of the children of node *N* are still locked by transaction *T*. Show that without this condition, the MGL protocol would be incorrect.

Database Recovery Techniques

In this chapter we discuss some of the techniques that can be used for database recovery from failures. In Section 21.1.4 we discussed the different causes of failure, such as system crashes

and transaction errors. Also, in Section 21.2, we covered many of the concepts that are used by recovery processes, such as the system log and commit points.

This chapter presents additional concepts that are relevant to recovery protocols, and provides an overview of the various database recovery algorithms We start in Section 23.1 with an outline of a typical recovery procedure and a categorization of recovery algorithms, and then we discuss several recovery concepts, including write-ahead logging, in-place versus shadow updates, and the process of rolling back (undoing) the effect of an incomplete or failed transaction. In Section 23.2 we pre-sent recovery techniques based on *deferred update*, also known as the NO-UNDO/REDO technique, where the data on disk is not updated until *after* a transaction commits. In Section 23.3 we discuss recovery techniques based on *immediate update*, where data can be updated on disk during transaction execution; these include the UNDO/REDO and UNDO/NO-REDO algorithms. We discuss the technique known as shadowing or shadow paging, which can be categorized as a NO-UNDO/NO-REDO algorithm in Section 23.4. An example of a practical DBMS recovery scheme, called ARIES, is presented in Section

23.5. Recovery in multidata-bases is briefly discussed in Section 23.6. Finally, techniques for recovery from cata-strophic failure are discussed in Section 23.7. Section 23.8 summarizes the chapter.

Our emphasis is on conceptually describing several different approaches to recov-ery. For descriptions of recovery features in specific systems, the reader should con-sult the bibliographic notes at the end of the chapter and the online and printed user manuals for those systems. Recovery techniques are often intertwined with the concurrency control mechanisms. Certain recovery techniques are best used with specific concurrency control methods. We will discuss recovery concepts indepen-dently of concurrency control mechanisms, but we will discuss the circumstances under which a particular recovery mechanism is best used with a certain concur-rency control protocol.

3.1 Recovery Concepts

3.1.1 Recovery Outline and Categorization of Recovery Algorithms

Recovery from transaction failures usually means that the database is *restored* to the most recent consistent state just before the time of failure. To do this, the system must keep information about the changes that were applied to data items by the various transactions. This information is typically kept in the **system log**, as we dis-cussed in Section 1.2.2. A typical strategy for recovery may be summarized infor-mally as follows:

If there is extensive damage to a wide portion of the database due to cata-strophic failure, such as a disk crash, the recovery method restores a past copy of the database that was *backed up* to archival storage (typically tape or other large capacity offline storage media) and reconstructs a more current state by reapplying or *redoing* the operations of committed transactions from the *backed up* log, up to the time of failure.

When the database on disk is not physically damaged, and a noncatastrophic failure of types 1 through 4 in Section 1.1.4 has occurred, the recovery strategy is to identify any changes that may cause an inconsistency in the database. For example, a transaction that has updated some database items on disk but has not been committed needs to have its changes reversed by *undoing* its write operations. It may also be necessary to *redo* some opera-tions in order to restore a consistent state of the database; for example, if a transaction has committed but some of its write operations have not yet been written to disk. For noncatastrophic failure, the recovery protocol does not need a complete archival copy of the database. Rather, the entries kept in the online system log on disk are analyzed to determine the appropriate actions for recovery.

Conceptually, we can distinguish two main techniques for recovery from noncata-strophic transaction failures: deferred update and immediate update. The **deferred update** techniques do not physically update the database on disk until *after* a trans-action reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transac-tion workspace or in the main memory buffers that the DBMS maintains (the DBMS main memory cache). Before commit, the updates are recorded persistently in the log, and then after commit, the updates are written to the database on disk. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet

have been recorded in the database on disk. Hence, deferred update is also known as the NO-UNDO/REDO **algorithm**.

In the **immediate update** techniques, the database *may be updated* by some opera-tions of a transaction *before* the transaction reaches its commit point. However, these operations must also be recorded in the log *on disk* by force-writing *before* they are applied to the database on disk, making recovery still possible. If a transaction fails after recording some changes in the database on disk but before reaching its commit point, the effect of its operations on the database must be undone; that is, the transaction must be rolled back. In the general case of immediate update, both *undo* and *redo* may be required during recovery. This technique, known as the UNDO/REDO **algorithm**, requires both operations during recovery, and is used most often in practice. A variation of the algorithm where all updates are required to be recorded in the database on disk *before* a transaction commits requires *undo* only, so it is known as the UNDO/NO-REDO **algorithm**. We discuss these techniques in Section 23.3.

The UNDO and REDO operations are required to be **idempotent**—that is, executing an operation multiple times is equivalent to executing it just once. In fact, the whole recovery process should be idempotent because if the system were to fail during the recovery process, the next recovery attempt might UNDO and REDO certain write_item operations that had already been executed during the first recovery process. The result of recovery from a system crash *during recovery* should be the same as the result of recovering *when there is no crash during recovery!*

3.1.2 Caching (Buffering) of Disk Blocks

The recovery process is often closely intertwined with operating system functions— in particular, the buffering of database disk pages in the DBMS main memory cache. Typically, multiple disk pages that include the data items to be updated are **cached** into main memory buffers and then updated in memory before being writ-ten back to disk. The caching of disk pages is traditionally an operating system func-tion, but because of its importance to the efficiency of recovery procedures, it is handled by the DBMS by calling low-level operating systems routines.

In general, it is convenient to consider recovery in terms of the database disk pages (blocks). Typically a collection of in-memory buffers, called the **DBMS cache**, is kept under the control of the DBMS for the purpose of holding these buffers. A **directory** for the cache is used to keep track of which database items are in the buffers.[1] This can be a table of <Disk_page_address, Buffer_location, ... > entries. When the DBMS requests action on some item, first it checks the cache directory to determine whether the disk page containing the item is in the DBMS cache. If it is not, the item must be located on disk, and the appropriate disk pages are copied into the cache. It may be necessary to **replace** (or **flush**) some of the cache buffers to make space available for the new item. Some page replacement strategy similar to these used in operating systems, such as least recently used (LRU) or first-in-first-out (FIFO), or a new strategy that is DBMS-specific can be used to select the buffers for replacement, such as DBMIN or Least-Likely-to-Use (see bibliographic notes).

The entries in the DBMS cache directory hold additional information relevant to buffer management. Associated with each buffer in the cache is a **dirty bit**, which can be included in the directory entry, to indicate whether or not the buffer has been modified. When a page is first read from the database disk into a cache buffer, a new entry is inserted in the cache directory with the new disk page address, and

the dirty bit is set to 0 (zero). As soon as the buffer is modified, the dirty bit for the corresponding directory entry is set to 1 (one). Additional information, such as the transaction id(s) of the transaction(s) that modified the buffer can also be kept in the directory. When the buffer contents are replaced (flushed) from the cache, the contents must first be written back to the corresponding disk page *only if its dirty bit is 1.* Another bit, called the **pin-unpin** bit, is also needed—a page in the cache is **pinned** (bit value 1 (one)) if it cannot be written back to disk as yet. For example, the recovery protocol may restrict certain buffer pages from being written back to the disk until the transactions that changed this buffer have committed.

Two main strategies can be employed when flushing a modified buffer back to disk. The first strategy, known as **in-place updating**, writes the buffer to the *same original disk location*, thus overwriting the old value of any changed data items on disk.[2] Hence, a single copy of each database disk block is maintained. The second strategy, known as **shadowing**, writes an updated buffer at a different disk location, so mul-tiple versions of data items can be maintained, but this approach is not typically used in practice.

In general, the old value of the data item before updating is called the **before image (BFIM**), and the new value after updating is called the **after image (AFIM**). If shad-owing is used, both the BFIM and the AFIM can be kept on disk; hence, it is not strictly necessary to maintain a log for recovering. We briefly discuss recovery based on shadowing in Section 23.4.

3.1.3 Write-Ahead Logging, Steal/No-Steal,and Force/No-Force

When in-place updating is used, it is necessary to use a log for recovery. In this case, the recovery mechanism must ensure that the BFIM of the data item is recorded in the appropriate log entry and that the log entry is flushed to disk before the BFIM is overwritten with the AFIM in the database on disk. This process is generally known as **write-ahead logging**, and is necessary to be able to UNDO the operation if this is required during recovery. Before we can describe a protocol for write-ahead logging, we need to distinguish between two types of log entry infor-mation included for a write command: the information needed for UNDO and the information needed for REDO. A REDO-**type log entry** includes the **new value** (AFIM) of the item written by the operation since this is needed to *redo* the effect of the operation from the log (by setting the item value in the database on disk to its AFIM). The UNDO-**type log entries** include the **old value** (BFIM) of the item since this is needed to *undo* the effect of the operation from the log (by setting the item value in the database back to its BFIM). In an UNDO/REDO algorithm, both types of log entries are combined. Additionally, when cascading rollback is possible, read_item entries in the log are considered to be UNDO-type entries .

As mentioned, the DBMS cache holds the cached database disk blocks in main memory buffers, which include not only *data blocks*, but also *index blocks* and *log blocks* from the disk. When a log record is written, it is stored in the current log buffer in the DBMS cache. The log is simply a sequential (append-only) disk file, and the DBMS cache may contain several log blocks in main memory buffers (typi-cally, the last *n* log blocks of the log file). When an update to a data block—stored in the DBMS cache—is made, an associated log record is written to the last log buffer in the DBMS cache. With the write-ahead logging approach, the log buffers (blocks) that contain the associated log records for a particular data block update *must first be written to disk* before the data block itself can be written back to disk from its main memory buffer.

Standard DBMS recovery terminology includes the terms **steal/no-steal** and **force/no-force**, which specify the rules that govern *when* a page from the database can be written to disk from the cache:

If a cache buffer page updated by a transaction *cannot* be written to disk before the transaction commits, the recovery method is called a **no-steal approach**. The pin-unpin bit will be used to indicate if a page cannot be written back to disk. On the other hand, if the recovery protocol allows writ-ing an updated buffer *before* the transaction commits, it is called **steal**. Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed. The *no-steal rule* means that UNDO will never be needed during recovery, since a commit-ted transaction will not have any of its updates on disk before it commits.

If all pages updated by a transaction are immediately written to disk *before* the transaction commits, it is called a **force approach**. Otherwise, it is called **no-force**. The *force rule* means that REDO will never be needed during recov-ery, since any committed transaction will have all its updates on disk before it is committed. page of a committed transaction may still be in the buffer when another transaction needs to update it, thus eliminating the I/O cost to write that page multiple times to disk, and possibly to have to read it again from disk. This may provide a substantial saving in the number of disk I/O operations when a specific page is updated heavily by multiple transactions.

To permit recovery when in-place updating is used, the appropriate entries required for recovery must be permanently recorded in the log on disk before changes are applied to the database. For example, consider the following **write-ahead logging (WAL**) protocol for a recovery algorithm that requires both UNDO and REDO:

The before image of an item cannot be overwritten by its after image in the database on disk until all UNDO-type log records for the updating transac-tion—up to this point—have been force-written to disk.

The commit operation of a transaction cannot be completed until all the REDO-type and UNDO-type log records for that transaction have been force-written to disk.

To facilitate the recovery process, the DBMS recovery subsystem may need to main-tain a number of lists related to the transactions being processed in the system. These include a list for **active transactions** that have started but not committed as yet, and it may also include lists of all **committed** and **aborted transactions** since the last checkpoint (see the next section). Maintaining these lists makes the recovery process more efficient.

3.1.4 Checkpoints in the System Log and Fuzzy Checkpointing

Another type of entry in the log is called a **checkpoint**.[3] A [checkpoint, *list of active transactions*] record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, *T*] entries in the log before a [checkpoint] entry do not need to have their WRITE operations

redone in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing. As part of checkpointing, the list of transaction ids for active transactions at the time of the checkpoint is included in the checkpoint record, so that these transactions can be easily identified during recovery.

The recovery manager of a DBMS must decide at what intervals to take a check-point. The interval may be measured in time—say, every *m* minutes—or in the number *t* of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters. Taking a checkpoint consists of the following actions:

Suspend execution of transactions temporarily.

Force-write all main memory buffers that have been modified to disk.

Write a [checkpoint] record to the log, and force-write the log to disk.

Resume executing transactions.

As a consequence of step 2, a checkpoint record in the log may also include addi-tional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active trans-action. This can facilitate undoing transaction operations in the event that a trans-action must be rolled back.

The time needed to force-write all modified memory buffers may delay transaction processing because of step 1. To reduce this delay, it is common to use a technique called **fuzzy checkpointing**. In this technique, the system can resume transaction processing after a [begin_checkpoint] record is written to the log without having to wait for step 2 to finish. When step 2 is completed, an [end_checkpoint, ...] record is written in the log with the relevant information collected during checkpointing. However, until step 2 is completed, the previous checkpoint record should remain valid. To accomplish this, the system maintains a file on disk that contains a pointer to the valid checkpoint, which continues to point to the previous checkpoint record in the log. Once step 2 is concluded, that pointer is changed to point to the new checkpoint in the log.

3.1.5 Transaction Rollback and Cascading Rollback

If a transaction fails for whatever reason after updating the database, but before the transaction commits, it may be necessary to **roll back** the transaction. If any data item values have been changed by the transaction and written to the database, they must be restored to their previous values (BFIMs). The undo-type log entries are used to restore the old values of data items that must be rolled back.

If a transaction *T* is rolled back, any transaction *S* that has, in the interim, read the value of some data item *X* written by *T* must also be rolled back. Similarly, once *S* is rolled back, any transaction *R* that has read the value of some data item *Y* written by *S* must also be rolled back; and so on. This phenomenon is

called **cascading roll-back**, and can occur when the recovery protocol ensures *recoverable* schedules but does not ensure *strict* or *cascadeless* schedules. Understandably, cascading rollback can be quite complex and time-consuming. That is why almost all recovery mechanisms are designed so that cascading rollback *is never required.*

Figure 3.1 shows an example where cascading rollback is required. The read and write operations of three individual transactions are shown in Figure 23.1(a). Figure 3.1(b) shows the system log at the point of a system crash for a particular execution schedule of these transactions. The values of data items *A*, *B*, *C*, and *D*, which are used by the transactions, are shown to the right of the system log entries. We assume that the original item values, shown in the first line, are $A = 30$, $B = 15$, $C = 40$, and $D = 20$. At the point of system failure, transaction T_3 has not reached its conclusion and must be rolled back. The WRITE operations of T_3, marked by a single * in Figure 3.1(b), are the T_3 operations that are undone during transaction rollback. Figure 3.1(c) graphically shows the operations of the different transactions along the time axis.

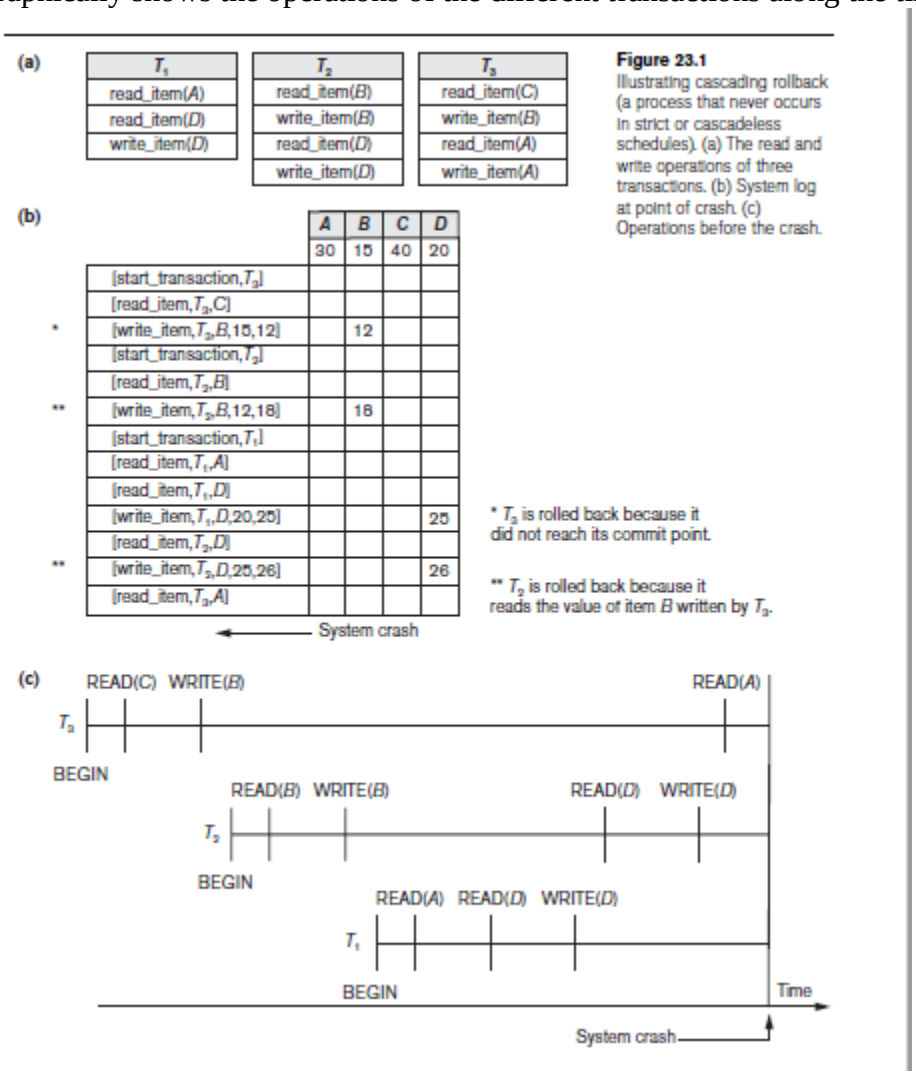


Figure 23.1
Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules). (a) The read and write operations of three transactions. (b) System log at point of crash. (c) Operations before the crash.

We must now check for cascading rollback. From Figure 23.1(c) we see that transac-tion T_2 reads the value of item *B* that was written by transaction T_3; this can also be determined by examining the log. Because T_3 is rolled back, T_2 must now be rolled back, too. The WRITE operations of T_2, marked by ** in the

log, are the ones that are undone. Note that only write_item operations need to be undone during transaction rollback; read_item operations are recorded in the log only to determine whether cascading rollback of additional transactions is necessary.

In practice, cascading rollback of transactions is *never* required because practical recovery methods *guarantee cascadeless or strict* schedules. Hence, there is also no need to record any read_item operations in the log because these are needed only for determining cascading rollback.

3.1.6 Transaction Actions That Do Not Affect the Database

In general, a transaction will have actions that do *not* affect the database, such as generating and printing messages or reports from information retrieved from the database. If a transaction fails before completion, we may not want the user to get these reports, since the transaction has failed to complete. If such erroneous reports are produced, part of the recovery process would have to inform the user that these reports are wrong, since the user may take an action based on these reports that affects the database. Hence, such reports should be generated only *after the transac-tion reaches its commit point.* A common method of dealing with such actions is to issue the commands that generate the reports but keep them as batch jobs, which are executed only after the transaction reaches its commit point. If the transaction fails, the batch jobs are canceled.

3.2 NO-UNDO/REDO Recovery Based on Deferred Update

The idea behind deferred update is to defer or postpone any actual updates to the database on disk until the transaction completes its execution successfully and reaches its commit point.[4]

During transaction execution, the updates are recorded only in the log and in the cache buffers. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database. If a transaction fails before reaching its commit point, there is no need to undo any operations because the transaction has not affected the database on disk in any way. Therefore, only REDO-**type log entries** are needed in the log, which include the **new value** (AFIM) of the item written by a write operation. The UNDO-**type log entries** are not needed since no undoing of operations will be required during recovery. Although this may sim-plify the recovery process, it cannot be used in practice unless transactions are short and each transaction changes few items. For other types of transactions, there is the potential for running out of buffer space because transaction changes must be held in the cache buffers until the commit point.

We can state a typical deferred update protocol as follows:

A transaction cannot change the database on disk until it reaches its commit point.

A transaction does not reach its commit point until all its REDO-type log entries are recorded in the log *and* the log buffer is force-written to disk.

Notice that step 2 of this protocol is a restatement of the write-ahead logging (WAL) protocol. Because the database is never updated on disk until after the transaction commits, there is never a need to UNDO any operations. REDO is needed in case the system fails after a transaction commits but before all its

changes are recorded in the database on disk. In this case, the transaction operations are redone from the log entries during recovery.

For multiuser systems with concurrency control, the concurrency control and recovery processes are interrelated. Consider a system in which concurrency control uses strict two-phase locking, so the locks on items remain in effect *until the trans-action reaches its commit point.* After that, the locks can be released. This ensures strict and serializable schedules. Assuming that [checkpoint] entries are included in the log, a possible recovery algorithm for this case, which we call RDU_M (Recovery using Deferred Update in a Multiuser environment), is given next.

Procedure RDU_M **(**NO-UNDO/REDO **with checkpoints).** Use two lists of transactions maintained by the system: the committed transactions *T* since the last checkpoint (**commit list**), and the active transactions *T* (**active list**). REDO all the WRITE operations of the committed transactions from the log, *in the order in which they were written into the log.* The transactions that are active and did not commit are effectively canceled and must be resubmitted.

The REDO procedure is defined as follows:

Procedure REDO (WRITE_OP)**.** Redoing a write_item operation WRITE_OP con-sists of examining its log entry [write_item, *T*, *X*, new_value] and setting the value of item *X* in the database to new_value, which is the after image (AFIM).

Figure 23.2 illustrates a timeline for a possible schedule of executing transactions. When the checkpoint was taken at time t_1, transaction T_1 had committed, whereas transactions T_3 and T_4 had not. Before the system crash at time t_2, T_3 and T_2 were committed but not T_4 and T_5. According to the RDU_M method, there is no need to redo the write_item operations of transaction T_1—or any transactions committed before the last checkpoint time t_1. The write_item operations of T_2 and T_3 must be redone, however, because both transactions reached their commit points after the last checkpoint. Recall that the log is force-written before committing a transaction. Transactions T_4 and T_5 are ignored: They are effectively canceled or rolled back because none of their write_item operations were recorded in the database on disk under the deferred update protocol.

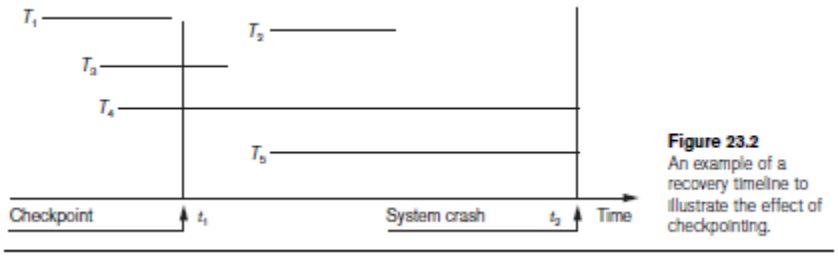


Figure 23.2
An example of a recovery timeline to illustrate the effect of checkpointing.

We can make the NO-UNDO/REDO recovery algorithm *more efficient* by noting that, if a data item *X* has been updated—as indicated in the log entries—more than once by committed transactions since the last checkpoint, it is only necessary to REDO *the last update of X* from the log during recovery because the other updates would be overwritten by this last REDO. In this case, we start from *the end of the log;* then, whenever an item is redone, it is added to a list of redone items. Before REDO is applied to an

item, the list is checked; if the item appears on the list, it is not redone again, since its last value has already been recovered.

If a transaction is aborted for any reason (say, by the deadlock detection method), it is simply resubmitted, since it has not changed the database on disk. A drawback of the method described here is that it limits the concurrent execution of transactions because *all write-locked items remain locked until the transaction reaches its commit point.* Additionally, it may require excessive buffer space to hold all updated items until the transactions commit. The method's main benefit is that transaction oper-ations *never need to be undone,* for two reasons:

A transaction does not record any changes in the database on disk until after it reaches its commit point—that is, until it completes its execution success-fully. Hence, a transaction is never rolled back because of failure during transaction execution.

A transaction will never read the value of an item that is written by an uncommitted transaction, because items remain locked until a transaction reaches its commit point. Hence, no cascading rollback will occur.

Figure 23.3 shows an example of recovery for a multiuser system that utilizes the recovery and concurrency control method just described.

3.3 Recovery Techniques Based on Immediate Update

In these techniques, when a transaction issues an update command, the database on disk can be updated *immediately*, without any need to wait for the transaction to reach its commit point. Notice that it is *not a requirement* that every update be applied immediately to disk; it is just possible that some updates are applied to disk *before the transaction commits.*

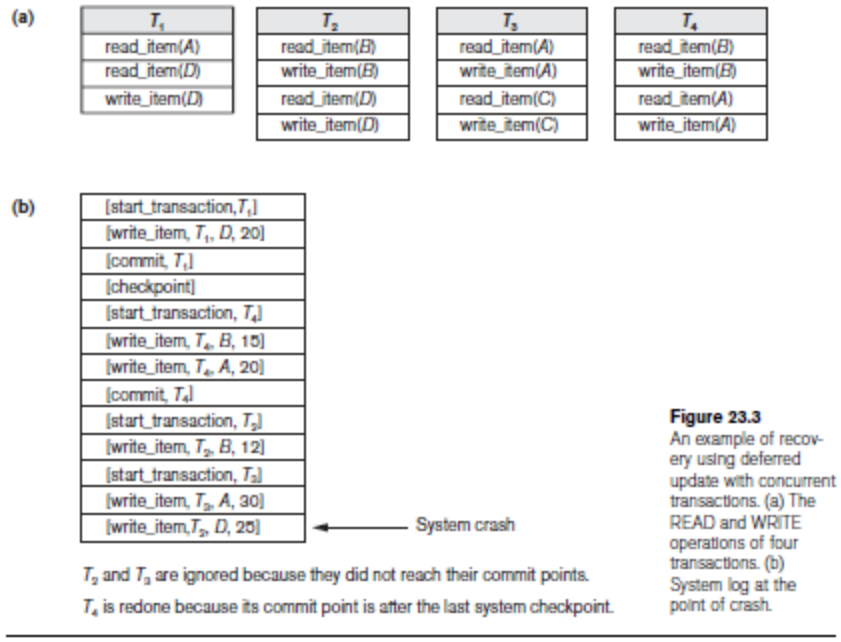


Figure 23.3
An example of recovery using deferred update with concurrent transactions. (a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

Provisions must be made for *undoing* the effect of update operations that have been applied to the database by a *failed transaction.* This is accomplished by rolling back the transaction and undoing the effect of the transaction's write_item operations. Therefore, the UNDO-**type log entries**, which include the **old value** (BFIM) of the item, must be stored in the log. Because UNDO can be needed during recovery, these methods follow a **steal strategy** for deciding when updated main memory buffers can be written back to disk (see Section 23.1.3). Theoretically, we can distinguish two main categories of immediate update algorithms. If the recovery technique ensures that all updates of a transaction are recorded in the database on disk *before the transaction commits*, there is never a need to REDO any operations of committed transactions. This is called the UNDO/NO-REDO **recovery algorithm**. In this method, all updates by a transaction must be recorded on disk *before the transaction commits*, so that REDO is never needed. Hence, this method must utilize the **force strategy** for deciding when updated main memory buffers are written back to disk.

If the transaction is allowed to commit before all its changes are written to the data-base, we have the most general case, known as the UNDO/REDO **recovery algo-rithm**. In this case, the **steal/no-force strategy** is applied (see Section 23.1.3). This is also the most complex technique. We will outline an UNDO/REDO recovery algo-rithm and leave it as an exercise for the reader to develop the UNDO/NO-REDO vari-ation. In Section 23.5, we describe a more practical approach known as the ARIES recovery technique.

When concurrent execution is permitted, the recovery process again depends on the protocols used for concurrency control. The procedure RIU_M (Recovery using Immediate Updates for a Multiuser environment) outlines a recovery algorithm for concurrent transactions with immediate update (UNDO/REDO recovery). Assume that the log includes checkpoints and that the concurrency control protocol pro-duces *strict schedules*—as, for example, the strict two-phase locking protocol does. Recall that a strict schedule does not allow a transaction to read or write an item unless the transaction that last

wrote the item has committed (or aborted and rolled back). However, deadlocks can occur in strict two-phase locking, thus requiring abort and UNDO of transactions. For a strict schedule, UNDO of an operation requires changing the item back to its old value (BFIM).

Procedure RIU_M **(**UNDO/REDO **with checkpoints).**

Use two lists of transactions maintained by the system: the committed trans-actions since the last checkpoint and the active transactions.

Undo all the write_item operations of the *active* (uncommitted) transactions, using the UNDO procedure. The operations should be undone in the reverse of the order in which they were written into the log.

Redo all the write_item operations of the *committed* transactions from the log, in the order in which they were written into the log, using the REDO proce-dure defined earlier.

The UNDO procedure is defined as follows:

Procedure UNDO **(**WRITE_OP**).** Undoing a write_item operation write_op con-sists of examining its log entry [write_item, *T*, *X*, old_value, new_value] and set-ting the value of item *X* in the database to old_value, which is the before image (BFIM). Undoing a number of write_item operations from one or more trans-actions from the log must proceed in the *reverse order* from the order in which the operations were written in the log.

As we discussed for the NO-UNDO/REDO procedure, step 3 is more efficiently done by starting from the *end of the log* and redoing only *the last update of each item X.* Whenever an item is redone, it is added to a list of redone items and is not redone again. A similar procedure can be devised to improve the efficiency of step 2 so that an item can be undone at most once during recovery. In this case, the earliest UNDO is applied first by scanning the log in the forward direction (starting from the beginning of the log). Whenever an item is undone, it is added to a list of undone items and is not undone again.

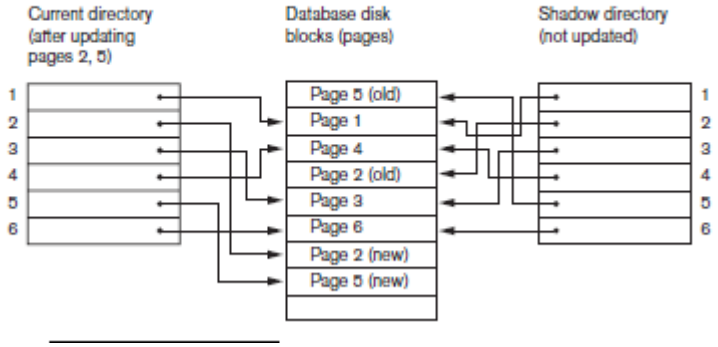
3.4 Shadow Paging

This recovery scheme does not require the use of a log in a single-user environment. In a multiuser environment, a log may be needed for the concurrency control method. Shadow paging considers the database to be made up of a number of fixed-size disk pages (or disk blocks)—say, *n*—for recovery purposes. A **directory** with *n* entries[5] is constructed, where the *i*th entry points to the *i*th database page on disk. The directory is kept in main memory if it is not too large, and all references—reads or writes—to database pages on disk go through it. When a transaction begins exe-cuting, the **current directory**—whose entries point to the most recent or current database pages on disk—is copied into a **shadow directory**. The shadow directory is then saved on disk while the current directory is used by the transaction.

During transaction execution, the shadow directory is *never* modified. When a write_item operation is performed, a new copy of the modified database page is cre-ated, but the old copy of that page is *not overwritten.* Instead, the new page is writ-ten elsewhere—on some previously unused disk block. The

current directory entry is modified to point to the new disk block, whereas the shadow directory is not modified and continues to point to the old unmodified disk block. Figure 23.4 illus-trates the concepts of shadow and current directories. For pages updated by the transaction, two versions are kept. The old version is referenced by the shadow directory and the new version by the current directory.

Figure 23.4
An example of shadow paging.



To recover from a failure during transaction execution, it is sufficient to free the modified database pages and to discard the current directory. The state of the data-base before transaction execution is available through the shadow directory, and that state is recovered by reinstating the shadow directory. The database thus is returned to its state prior to the transaction that was executing when the crash occurred, and any modified pages are discarded. Committing a transaction corre-sponds to discarding the previous shadow directory. Since recovery involves neither undoing nor redoing data items, this technique can be categorized as a NO-UNDO/NO-REDO technique for recovery.

In a multiuser environment with concurrent transactions, logs and checkpoints must be incorporated into the shadow paging technique. One disadvantage of shadow paging is that the updated database pages change location on disk. This makes it dif-ficult to keep related database pages close together on disk without complex storage management strategies. Furthermore, if the directory is large, the overhead of writ-ing shadow directories to disk as transactions commit is significant. A further com-plication is how to handle **garbage collection** when a transaction commits. The old pages referenced by the shadow directory that have been updated must be released and added to a list of free pages for future use. These pages are no longer needed after the transaction commits. Another issue is that the operation to migrate between cur-rent and shadow directories must be implemented as an atomic operation.

3.5 The ARIES Recovery Algorithm

We now describe the ARIES algorithm as an example of a recovery algorithm used in database systems. It is used in many relational database-related products of IBM. ARIES uses a steal/no-force approach for writing, and it is based on three concepts: write-ahead logging, repeating history during redo, and logging changes during undo. We discussed write-ahead logging in Section 23.1.3. The second concept, **repeating history**, means that ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state *when the crash occurred.* Transactions that were uncommitted at the time of the crash (active transactions) are undone. The third concept, **logging during undo**, will prevent

ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

The ARIES recovery procedure consists of three main steps: analysis, REDO, and UNDO. The **analysis step** identifies the dirty (updated) pages in the buffer[6] and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The REDO **phase** actu-ally reapplies updates from the log to the database. Generally, the REDO operation is applied only to committed transactions. However, this is not the case in ARIES. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. Additionally, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and therefore does not need to be reapplied. Thus, *only the necessary REDO operations* are applied during recovery. Finally, during the UNDO **phase**, the log is scanned backward and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. Additionally, checkpointing is used. These tables are maintained by the trans-action manager and written to the log during checkpointing.

In ARIES, every log record has an associated **log sequence number (LSN)** that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. Also, each data page will store the LSN of the *latest log record corresponding to a change for that page.* A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log. When a transaction ends, whether by committing or aborting, an *end log record* is written.

Common fields in all log records include the previous LSN for that transaction, the transaction ID, and the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write) action, additional fields in the log record include the page ID for the page that con-tains the item, the length of the updated item, its offset from the beginning of the page, the before image of the item, and its after image.

Besides the log, two tables are needed for efficient recovery: the **Transaction Table** and the **Dirty Page Table**, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction*, with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

Checkpointing in ARIES consists of the following: writing a begin_checkpoint record to the log, writing an end_checkpoint record to the log, and writing *the LSN of* the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the end_checkpoint record, the con-tents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, **fuzzy checkpointing** is used so that the DBMS can con-tinue to execute transactions during checkpointing (see Section 23.1.4). Additionally, the contents

of the DBMS cache do not have to be flushed to disk dur-ing checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery. Note

that if a crash occurs during checkpointing, the special file will refer to the previous checkpoint, which is used for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The **analysis phase** starts at the begin_checkpoint record and proceeds to the end of the log. When the end_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction *T* in the Transaction Table, then the entry for *T* is deleted from that table. If some other type of log record is encountered for a transaction *T* , then an entry for *T* is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page *P*, then an entry would be made for page *P* (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The REDO **phase** follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk*. It can determine this by finding the smallest LSN, *M*, of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to an LSN < *M*, for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = *M* and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page *P* that is not in the Dirty Page Table, then this change is already on disk and does not need to be reapplied. Or, if a change recorded in the log (with LSN = *N*, say) pertains to page *P* and the Dirty Page Table contains an entry for *P* with LSN greater than *N*, then the change is already present. If neither of these two conditions hold, page *P* is read from disk and the LSN stored on that page, LSN(*P*), is compared with *N*. If *N* < LSN(*P*), then the change has been applied and the page does not need to be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the undo_set—has been identified in the Transaction Table during the analysis phase. Now, the UNDO **phase** proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of trans-actions in the undo_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Consider the recovery example shown in Figure 23.5. There are three transactions:
T_1, T_2, and T_3. T_1 updates page *C*, T_2 updates pages *B* and *C*, and T_3 updates page A.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure 23.5
An example of recovery in ARIES. (a) The log at point of crash. (b) The Transaction and Dirty Page Tables at time of checkpoint. (c) The Transaction and Dirty Page Tables after the analysis phase.

Figure 23.5(a) shows the partial contents of the log, and Figure 23.5(b) shows the contents of the Transaction Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end_checkpoint record would contain the Transaction Table and Dirty Page Table in Figure 23.5(b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction T_3 is made in the Transaction Table and a new entry for page A is made in the Dirty Page Table. After log record 8 is analyzed, the status of transaction T_2 is changed to committed in the Transaction Table. Figure 23.5(c) shows the two tables after the analysis phase. For the REDO phase, the smallest LSN in the Dirty Page Table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages C, B, A, and C, respectively, are not less than the LSNs of those pages (as shown in the Dirty Page Table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less then the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure 23.5(c)), UNDO is applied only to the active transaction T_3. The UNDO phase starts at log entry 6 (the last update for T_3) and proceeds backward in the log. The backward chain of updates for transaction T_3 (only log record 6 in this exam-ple) is followed and undone.

3.6 Recovery in Multidatabase Systems

So far, we have implicitly assumed that a transaction accesses a single database. In some cases, a single transaction, called a **multidatabase transaction**, may require access to multiple databases. These databases may even be stored on different types of DBMSs; for example, some DBMSs may be relational, whereas others are object-oriented, hierarchical, or network DBMSs. In such a case, each DBMS involved in the multidatabase transaction may have its own recovery technique and transaction manager separate from those of the other DBMSs. This situation is somewhat simi-lar to the case of a distributed database management system, where parts of the database reside at different sites that are connected by a communication network.

To maintain the atomicity of a multidatabase transaction, it is necessary to have a two-level recovery mechanism. A **global recovery manager**, or **coordinator**, is needed to maintain information needed for recovery, in addition to the local recov-ery managers and the information they maintain (log, tables). The coordinator usu-ally follows a protocol called the **two-phase commit protocol**, whose two phases can be stated as follows:

Phase 1. When all participating databases signal the coordinator that the part of the multidatabase transaction involving each has concluded, the coordinator sends a message *prepare for commit* to each participant to get ready for committing the transaction. Each participating database receiving that message will force-write all log records and needed information for local recovery to disk and then send a *ready to commit* or *OK* signal to the coordinator. If the force-writing to disk fails or the local transaction cannot commit for some reason, the participating database sends a *cannot commit* or *not OK* signal to the coordinator. If the coordinator does not receive a reply from the database within a certain time out interval, it assumes a *not OK* response.

Phase 2. If *all* participating databases reply *OK*, and the coordinator's vote is also *OK*, the transaction is successful, and the coordinator sends a *commit* signal for the transaction to the participating databases. Because all the local effects of the transaction and information needed for local recovery have been recorded in the logs of the participating databases, recovery from fail-ure is now possible. Each participating database completes transaction com-mit by writing a [commit] entry for the transaction in the log and permanently updating the database if needed. On the other hand, if one or more of the participating databases or the coordinator have a *not OK* response, the transaction has failed, and the coordinator sends a message to *roll back* or UNDO the local effect of the transaction to each participating database. This is done by undoing the transaction operations, using the log.

The net effect of the two-phase commit protocol is that either all participating data-bases commit the effect of the transaction or none of them do. In case any of the participants—or the coordinator—fails, it is always possible to recover to a state where either the transaction is committed or it is rolled back. A failure during or before Phase 1 usually requires the transaction to be rolled back, whereas a failure during Phase 2 means that a successful transaction can recover and commit.

3.7 Database Backup and Recovery from Catastrophic Failures

So far, all the techniques we have discussed apply to noncatastrophic failures. A key assumption has been that the system log is maintained on the disk and is not lost as a result of the failure. Similarly, the

shadow directory must be stored on disk to allow recovery when shadow paging is used. The recovery techniques we have dis-cussed use the entries in the system log or the shadow directory to recover from fail-ure by bringing the database back to a consistent state.

The recovery manager of a DBMS must also be equipped to handle more cata-strophic failures such as disk crashes. The main technique used to handle such crashes is a **database backup**, in which the whole database and the log are periodi-cally copied onto a cheap storage medium such as magnetic tapes or other large capacity offline storage devices. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.

Data from critical applications such as banking, insurance, stock market, and other databases is periodically backed up in its entirety and moved to physically separate safe locations. Subterranean storage vaults have been used to protect such data from flood, storm, earthquake, or fire damage. Events like the 9/11 terrorist attack in New York (in 2001) and the Katrina hurricane disaster in New Orleans (in 2005) have created a greater awareness of *disaster recovery of business-critical databases*.

To avoid losing all the effects of transactions that have been executed since the last backup, it is customary to back up the system log at more frequent intervals than full database backup by periodically copying it to magnetic tape. The system log is usually substantially smaller than the database itself and hence can be backed up more frequently. Therefore, users do not lose all transactions they have performed

since the last database backup. All committed transactions recorded in the portion of the system log that has been backed up to tape can have their effect on the data-base redone. A new log is started after each database backup. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape. Following that, the effects of all the committed transactions whose operations have been recorded in the backed-up copies of the system log are reconstructed.

3.8 Summary

In this chapter we discussed the techniques for recovery from transaction failures. The main goal of recovery is to ensure the atomicity property of a transaction. If a transaction fails before completing its execution, the recovery mechanism has to make sure that the transaction has no lasting effects on the database. First we gave an informal outline for a recovery process and then we discussed system concepts for recovery. These included a discussion of caching, in-place updating versus shad-owing, before and after images of a data item, UNDO versus REDO recovery opera-tions, steal/no-steal and force/no-force policies, system checkpointing, and the write-ahead logging protocol.

Next we discussed two different approaches to recovery: deferred update and imme-diate update. Deferred update techniques postpone any actual updating of the data-base on disk until a transaction reaches its commit point. The transaction force-writes the log to disk before recording the updates in the database. This approach, when used with certain concurrency control methods, is designed never to require transaction rollback, and recovery simply consists of redoing the opera-tions of transactions committed

after the last checkpoint from the log. The disad-vantage is that too much buffer space may be needed, since updates are kept in the buffers and are not applied to disk until a transaction commits. Deferred update can lead to a recovery algorithm known as NO-UNDO/REDO. Immediate update tech-niques may apply changes to the database on disk before the transaction reaches a successful conclusion. Any changes applied to the database must first be recorded in the log and force-written to disk so that these operations can be undone if neces-sary. We also gave an overview of a recovery algorithm for immediate update known as UNDO/REDO. Another algorithm, known as UNDO/NO-REDO, can also be devel-oped for immediate update if all transaction actions are recorded in the database before commit.

We discussed the shadow paging technique for recovery, which keeps track of old database pages by using a shadow directory. This technique, which is classified as NO-UNDO/NO-REDO, does not require a log in single-user systems but still needs the log for multiuser systems. We also presented ARIES, a specific recovery scheme used in many of IBM's relational database products. Then we discussed the two-phase commit protocol, which is used for recovery from failures involving multi-database transactions. Finally, we discussed recovery from catastrophic failures, which is typically done by backing up the database and the log to tape. The log can be backed up more frequently than the database, and the backup log can be used to redo operations starting from the last database backup.

Review Questions

3.1. Discuss the different types of transaction failures. What is meant by cata-strophic failure?

3.2. Discuss the actions taken by the read_item and write_item operations on a database.

3.3. What is the system log used for? What are the typical kinds of entries in a system log? What are checkpoints, and why are they important? What are transaction commit points, and why are they important?

3.4. How are buffering and caching techniques used by the recovery subsystem?

3.5. What are the before image (BFIM) and after image (AFIM) of a data item? What is the difference between in-place updating and shadowing, with respect to their handling of BFIM and AFIM?

3.6. What are UNDO-type and REDO-type log entries?

3.7. Describe the write-ahead logging protocol.

3.8. Identify three typical lists of transactions that are maintained by the recovery subsystem.

3.9. What is meant by transaction rollback? What is meant by cascading rollback? Why do practical recovery methods use protocols that do not permit cascad-ing rollback? Which recovery techniques do not require any rollback?

3.10. Discuss the UNDO and REDO operations and the recovery techniques that use each.

3.11. Discuss the deferred update technique of recovery. What are the advantages and disadvantages of this technique? Why is it called the NO-UNDO/REDO method?

3.12. How can recovery handle transaction operations that do not affect the data-base, such as the printing of reports by a transaction?

3.13. Discuss the immediate update recovery technique in both single-user and multiuser environments. What are the advantages and disadvantages of immediate update?

3.14. What is the difference between the UNDO/REDO and the UNDO/NO-REDO algorithms for recovery with immediate update? Develop the outline for an UNDO/NO-REDO algorithm.

3.15. Describe the shadow paging recovery technique. Under what circumstances does it not require a log?

3.16. Describe the three phases of the ARIES recovery method.

.17. What are log sequence numbers (LSNs) in ARIES? How are they used? What information do the Dirty Page Table and Transaction Table contain? Describe how fuzzy checkpointing is used in ARIES.

3.18. What do the terms steal/no-steal and force/no-force mean with regard to buffer management for transaction processing?

3.19. Describe the two-phase commit protocol for multidatabase transactions.

3.20. Discuss how disaster recovery from catastrophic failures is handled.

Exercises

3.21. Suppose that the system crashes before the [read_item, T_3, A] entry is written to the log in Figure 23.1(b). Will that make any difference in the recovery process?

3.22. Suppose that the system crashes before the [write_item, T_2, D, 25, 26] entry is written to the log in Figure 23.1(b). Will that make any difference in the recovery process?

3.23. Figure 23.6 shows the log corresponding to a particular schedule at the point of a system crash for four transactions T_1, T_2, T_3, and T_4. Suppose that we use the *immediate update protocol* with checkpointing. Describe the recovery process from the system crash. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone, and

whether any cascading rollback takes place.

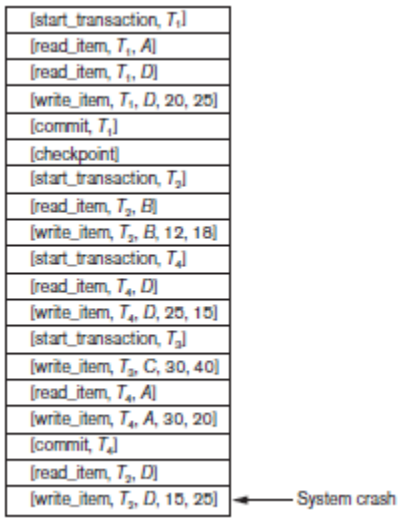


Figure 23.6
A sample schedule and its corresponding log.

3.24. Suppose that we use the deferred update protocol for the example in Figure 23.6. Show how the log would be different in the case of deferred update by removing the unnecessary log entries; then describe the recovery process, using your modified log. Assume that only REDO operations are applied, and specify which operations in the log are redone and which are ignored.

3.25. How does checkpointing in ARIES differ from checkpointing as described in Section 23.1.4?

3.26. How are log sequence numbers used by ARIES to reduce the amount of REDO work needed for recovery? Illustrate with an example using the infor-mation shown in Figure 23.5. You can make your own assumptions as to when a page is written to disk.

3.27. What implications would a no-steal/force buffer management policy have on checkpointing and recovery?

Choose the correct answer for each of the following multiple-choice questions:

3.28. Incremental logging with deferred updates implies that the recovery system must necessarily

store the old value of the updated item in the log. store the new value of the updated item in the log. store both the old and new value of the updated item in the log. store only the Begin Transaction and Commit Transaction records in the log.

3.29. The write-ahead logging (WAL) protocol simply means that writing of a data item should be done ahead of any logging operation. the log record for an operation should be written before the actual data is written. all log records should be written before a new transaction begins execu-tion. the log never needs to be written to disk.

3.30. In case of transaction failure under a deferred update incremental logging scheme, which of the following will be needed? an undo operation a redo operation an undo and redo operation none of the above

3.31. For incremental logging with immediate updates, a log record for a transac-tion would contain

a transaction name, a data item name, and the old and new value of the item.a transaction name, a data item name, and the old value of the item.a transaction name, a data item name, and the new value of the item.a transaction name and a data item name.

3.32. For correct behavior during recovery, undo and redo operations must be commutative. associative.
idempotent. distributive.

3.33. When a failure occurs, the log is consulted and each operation is either undone or redone. This is a problem because searching the entire log is time consuming. many redos are unnecessary. both (a) and (b). none of the above.

3.34. When using a log-based recovery scheme, it might improve performance as well as providing a recovery mechanism by writing the log records to disk when each transaction commits. writing the appropriate log records to disk during the transaction's execu-tion. waiting to write the log records until multiple transactions commit and writing them as a batch. never writing the log records to disk.

3.35. There is a possibility of a cascading rollback when

a. a transaction writes items that have been written only by a committed transaction. a transaction writes an item that is previously written by an uncommitted transaction. a transaction reads an item that is previously written by an uncommitted transaction. both (b) and (c).

3.36. To cope with media (disk) failures, it is necessary for the DBMS to only execute transactions in a single user environment. to keep a redundant copy of the database. to never abort a transaction.all of the above.